Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**

Grado en Ingeniería Informática

Trabajo Fin de Grado

Conway's Game of Life

Autor: Guillermo López García
Tutor(a): Jonathan Sánchez Hernández

Madrid, 06 - 2022

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática

Título: Conway's Game of Life

06 - 2022

Autor: Guillermo López García
Tutor: Jonathan Sánchez Hernández
Departamento de Matemática Aplicada a las TIC
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

Una de las ideas más equivocadas es que los comportamientos complejos deben surgir necesariamente de sistemas igual de complejos. Incluso John von Neumann cayó en la misma confusión al intentar averiguar una forma de imitar uno de los fenómenos más fundamentales y sofisticados de los seres vivos, la autorreplicación celular. Su objetivo era definir una máquina capaz de generar copias de sí misma, concibiendo de este modo el término de autómata celular. Pese a cumplir con su propósito, su solución no suscitó mucho interés precisamente por la complejidad de la misma.

No fue hasta décadas más tarde cuando John Horton Conway demostró que, de hecho, partiendo de un conjunto sencillo de reglas se pueden llegar a imitar comportamientos de lo más complejos. En concreto, su publicación del juego matemático denominado El Juego de la Vida, mostró cómo con un tablero análogo al de Go y 4 normas se podía construir un ordenador capaz de computar cualquier programa. Como no podía ser de otra forma, obtuvo una gran popularidad desde un inicio cautivando a la comunidad hasta día de hoy.

El siguiente documento sirve como estudio del término de autómatas celulares partiendo desde una introducción al tema hasta llegar al ya mencionado Juego de la Vida de Conway. En los siguientes capítulos se mostrarán los patrones que le dan la capacidad de computación, así como una implementación sencilla en Python. Finalmente, se estudiará el desarrollo de un algoritmo genético capaz de encontrar nuevas configuraciones, que, aunque distintas al Juego de la Vida, intenten imitar su comportamiento.

Abstract

One of the most common misbeliefs is that complex behaviours should necessarilly arise from systems just as complex. Even John von Neumann made the same confussion when trying to imitate one of the most fundamental and sophisticated phenomena of living beings, cellular self-replication. His objective was to define a machine capable of making exact copies of itself and thus giving birth to the term cellular automaton. Despite achieving his main goal, his solution did not arouse great curiosity precisely due to its complexity.

It was not until decades later when John Horton Conway proved that, in fact, just with a simple set of rules one could imitate truly complex behaviours. In particular, with his release of the mathemathical game named Game of Life, he showed that a Go board and 4 simple rules were enough to build a computer able to run any program. As it would be expected, the game caused great popularity captivating the community to this day.

The following document studies the term cellular automaton, starting from a brief introduction and background history up to the already mentioned Game of Life. On the following chapters, the patterns that give them the ability of computability will be shown, as well of a simple implementation of the game in Python. Lastly, it will be described the development of a Genetic Algorithm capable of finding new configurations that, although different from the one in Game of Life, share some of its behaviours.

Tabla de contenidos

1. Introducción **1**

2. Autómatas Celulares **3**
2.1. Introducción a los Autómatas Celulares . . . 3
2.2. Breve historia . . . 3

3. El Juego de la Vida de Conway **7**
3.1. Descripción y reglas . . . 7
3.2. Clasificación de patrones . . . 8
3.2.1. *Oscillators* . . . 8
3.2.2. *Still lifes* . . . 9
3.2.3. *Spaceships* . . . 9
3.2.4. Configuraciones complejas . . . 10
3.3. Definición de autómata celular abstracto . . . 12
3.4. Implementación . . . 14
3.4.1. Separación en módulos . . . 15
3.4.2. universe.py . . . 15
3.4.3. game.py . . . 15
3.4.4. Controles del programa . . . 18
3.4.5. Gestión de los datos . . . 18
3.4.6. Ejemplos de ejecución . . . 19

4. Algoritmo Genético **23**
4.1. Introducción a los algoritmos evolutivos . . . 23
4.2. Objetivos y diseño del algoritmo. . . . 24
4.2.1. Población inicial . . . 25
4.2.2. Función objetivo (*Fitness*) . . . 26
4.2.3. Selección (*Selection*) . . . 27
4.2.4. Recombinación (*Crossover*) . . . 27
4.2.5. Mutación (*Mutation*) . . . 27
4.2.6. Condición de parada . . . 28
4.3. Ayudas para la implementación . . . 28
4.3.1. Gestión de la base de datos . . . 28
4.3.2. Clase *Rule()* . . . 29
4.4. Implementación . . . 30
4.4.1. crossover.py . . . 31

4.4.2. fitness.py y simulation.py . . . 31
4.4.3. mutation.py . . . 33
4.4.4. selection.py . . . 34
4.4.5. main.py y init.py . . . 34
4.5. Ejecución del algoritmo y representación gráfica de reglas . . . 38

5. Resultados **41**
5.1. Regla inicial 12/33 . . . 41
5.1.1. Resultado 1 . . . 41
5.1.1.1. *Still lifes* . . . 41
5.1.1.2. *Oscillators* . . . 42
5.1.1.3. *Spaceships* . . . 42
5.1.2. Resultado 2 . . . 43
5.1.2.1. *Still lifes* . . . 43
5.1.2.2. *Oscillators* . . . 43
5.1.2.3. *Spaceships* . . . 43
5.2. Regla inicial 23/44 . . . 43
5.2.1. Resultado 3 . . . 43
5.2.1.1. *Still lifes* . . . 44
5.2.1.2. *Oscillators* . . . 44
5.2.1.3. *Spaceships* . . . 44
5.3. Regla inicial 24/44 . . . 45
5.3.1. Resultado 4 . . . 45
5.3.1.1. *Still lifes* . . . 45
5.3.1.2. *Oscillators* . . . 45
5.3.1.3. *Spaceships* . . . 46
5.3.2. Resultado 5 . . . 46
5.3.2.1. *Still lifes* . . . 47
5.3.2.2. *Oscillators* . . . 47
5.3.2.3. *Spaceships* . . . 47
5.4. Conclusiones y posibles mejoras . . . 48

6. Análisis de impacto **51**

Bibliografía **53**

Capítulo 1

Introducción

A lo largo de los siguientes capítulos se muestra la memoria del proyecto realizado en torno a el Juego de la Vida de Conway. En un principio, el desarrollo de otras versiones del juego (con celdas hexagonales, cubos en tres dimensiones, etc) iba a ser el tema central del trabajo, pero finalmente se decidió enfocarlo en las características y capacidades que se pueden encontrar en los autómatas celulares binarios de células cuadradas. Además de ser un tema más completo, esta decisión permitía hacer una conexión con los algoritmos evolutivos y cómo éstos pueden ayudarnos en el estudio de los autómata celulares. En concreto, la memoria explicativa del trabajo se ha dividido en seis capítulos incluyendo este a modo de introducción.

En primer lugar sin contar la introducción, el capítulo 2: Autómatas Celulares. Como su nombre indica, se trata de una breve introducción al término de los autómatas celulares. Además se incluye una descripción de su contexto histórico.

En segundo lugar, el capítulo 3: El Juego de la Vida de Conway. Este capítulo está dedicado concretamente al juego matemático creado por John Horton Conway. Se empieza con una definición del mismo y sus reglas en el contexto de los autómatas celulares, seguido por una clasificación de los patrones más comunes y otras estructuras complejas que se pueden formar. También se incluye una descripción de un autómata celular abstracto y, por último, se explica el desarrollo de una implementación del Juego de la Vida en Python.

A continuación, el capítulo 4: Algoritmo Genético. Una vez visto el Juego de la Vida y su capacidad de imitar comportamientos complejos, se muestra el diseño e implementación de un algoritmo que busque nuevas reglas con la misma capacidad. Para ello se presenta el término de algoritmos evolutivos, un esquema de su funcionamiento y los operadores que debe contener. El diseño del algoritmo está basado en el artículo *Research of a Cellular Automaton Simulating Logic Gates by Evolutionary Algorithms* [1].

El capítulo 5: Resultados. En este apartado se incluyen los resultados del algoritmo desarrollado en el capítulo anterior. Se muestra una colección de reglas resultantes y una representación gráfica de los patrones encontrados, poniendo especial interés en los patrones dinámicos.

En último lugar, capítulo 6: Análisis de impacto. Sirve como una breve reflexión sobre los impactos del trabajo realizado y el alcance de sus beneficios.

Capítulo 2

Autómatas Celulares

Antes de comenzar con la explicación del Juego de la Vida de Conway, es necesario entender qué son los Autómatas Celulares. En este capítulo, se muestra una breve introducción al término, partiendo desde una descripción más informal y explicación sobre sus orígenes.

2.1. Introducción a los Autómatas Celulares

La forma más sencilla de entender el concepto es imaginarse una rejilla, finita o infinita, de elementos llamados *células*. Cada una de ellas tiene un estado que varía con el tiempo y que depende particularmente de las células más cercanas. En cada instante de tiempo, mirarán la disposición de sus vecinas y, teniendo en cuenta una regla de transición idéntica en toda la rejilla, tomarán una decisión para saber a qué estado deben evolucionar.

Normalmente se suele identificar a cada célula como un cuadrado, formando en su conjunto una gran cuadrícula como es en el caso del Juego de la Vida. Sin embargo, existen autómatas celulares formados por hexágonos [2], formas irregulares [3], e incluso cubos en tres dimensiones [4].

Al contrario de lo que pudiese parecer, cuentan con una gran cantidad de usos entre los que cabe destacar: modelos sobre el tráfico en carreteras [5], sobre la transmisión de ciertas enfermedades [6], o simulaciones sobre la expansión de incendios forestales [7]. No obstante, más allá de sus aplicaciones, uno de los aspectos más llamativos es que son capaces generar patrones o estructuras complejas partiendo de reglas sencillas. Desde patrones similares a copos de nieve o modelos computacionales, hasta el caso del autómata celular que forma diseños muy parecidos a los de conchas de ciertos moluscos [8].

Este último ejemplo suscita gran interés ya que, precisamente, el origen de los autómatas celulares está fuertemente relacionado con imitar comportamientos procedentes de la biología.

2.2. Breve historia

A finales de los años 40, comienzos de los 50, John von Neumann inventó el concepto de los autómatas celulares [9, Capítulo 25]. Su objetivo principal era construir una máquina

capaz de autorreplicarse imitando al comportamiento habitual de las células en los seres vivos. Desde un primer momento se pensó que dada la complejidad del funcionamiento de los ordenadores y los organismos biológicos, algo como la autorreplicación debía ser procedente necesariamente de un sistema igual de complejo.

De este modo nació el primer autómata celular: una rejilla formada por elementos cuadrados en el que cada una de las células podía tomar 29 estados diferentes. Cada célula dependía de sus vecinas adyacentes únicamente, creando así el término de *vecindario de Neumann*. Como se ve en la figura 2.1, el estado de la célula central viene dado únicamente por las vecinas adyacentes a sus lados, sin contar las diagonales. Esta configuración de reglas era, efectivamente, capaz de generar un estructura que crecía con el tiempo generando el mismo patrón continuamente. No obstante, el estado de la tecnología en esa época y el hecho de ser un modelo complicado provocó una escasez de investigaciones científicas en los siguientes años.

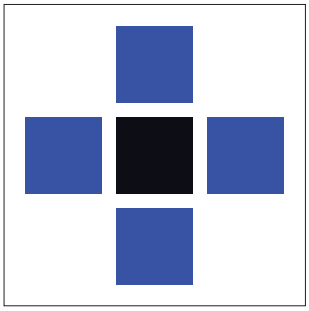

Figura 2.1: Vecindario de Neumann.

Décadas más tarde, en octubre de 1970, se presentó el Juego de la Vida en la revista matemática de *Mathematical games* publicado por Martin Gardner y desarrollado por John Horton Conway. Se trataba de un autómata celular compuesto también por células cuadradas, pero que, a diferencia del autómata de Neumann, éste contaba solo con 2 estados. Desde el comienzo causó gran interés en el público dada la sencillez del juego y su capacidad de generar patrones complejos, convirtiéndose así en uno de los artículos más leídos del semanario. Pronto los lectores empezaron a documentar nuevas estructuras, incluida una de las más importantes: las *Gosper Glider Gun* denominadas así por su creador, Bill Gosper. Dicho patrón era capaz de crecer indefinidamente, demostrando que la autorreplicación no es obligatoriamente consecuencia de reglas complejas. En el capítulo 3 se muestra en más detalle las implicaciones de dicha estructura y la capacidad de computación del Juego de la Vida.

Desde ese momento en adelante se retomó la investigación en este campo, pudiendo destacar la labor de Stephen Wolfram. A partir de la década de 1980, trabajó con autómatas celulares para realizar diferentes simulaciones físicas o crear la máquina de Turing más pequeña. Por otra parte, publicó en su libro *A New Kind of Science* [10] el primer intento de clasificación de autómatas según su comportamiento y patrones generados. La clasificación consta de 4 clases o categorías:

- *Clase 1*: Desde un estado inicial aleatorio las células evolucionan rápidamente al mismo estado, tendiendo a una estabilidad en toda la rejilla.

- *Clase 2*: En este caso, las células tienden a formar estructuras estables o patrones estáticos que se repiten con el tiempo. Ver 3.2.1.
- *Clase 3*: Las células de la rejilla rápidamente forman estructuras aleatorias y crecientes, destruyendo los posibles patrones estables iniciales. Son de naturaleza caótica.
- *Clase 4*: Se trata de la categoría más interesante y menos común de todas. Los autómatas de esta clase tienden a formar estructuras complejas que son capaces de sobrevivir por grandes periodos de tiempo, pudiendo aparentar un movimiento a través de la rejilla. Después de un periodo de tiempo imposible de determinar desde un estado inicial, pueden evolucionar a un estado similar al de los de Clase 2 en el que todos los patrones son estables y estáticos. Dentro de esta categoría entraría el Juego de la Vida de Conway.

Capítulo 3

El Juego de la Vida de Conway

3.1. Descripción y reglas

El Juego de la Vida de Conway se trata de un autómata celular con teselación cuadrada, en la que las células o celdas pueden tomar dos posibles estados, viva o muerta. Cada una de ellas cambiará su estado respecto a un tiempo discreto y teniendo en cuenta el estado actual de sus células vecinas (las más cercanas). En este caso, el siguiente estado de cada célula vendrá dado por el estado de su *vecindario de Moore* (figura 3.1), es decir, exactamente de sus 8 células adyacentes. Al conjunto de todas las celdas y sus vecindarios se le denomina como *universo* (figura 3.2). Tanto esta última figura como la 3.3, 3.5, 3.6, 3.8 y 3.9 han sido generadas por una implementación online del Juego de la Vida [11].

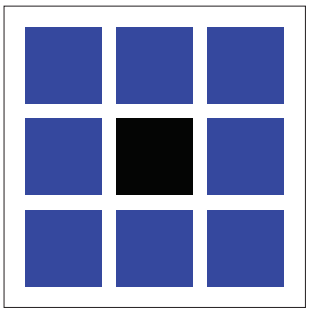

Figura 3.1: Vecindario de Moore

Una vez conocido el vecindario, el siguiente estado de nuestra celda está determinado por únicamente 4 reglas:

- Una célula **viva** seguirá viva si tiene 2 ó 3 vecinas vivas.
- Una célula **viva** morirá por sobrepoblación si tiene más de 3 vecinas vivas.
- Una célula **viva** morirá por soledad si tiene menos de 2 vecinas vivas.
- Una célula **muerta** revivirá si tiene exactamente 3 vecinas vivas.

A pesar de la simplicidad que aparenta, lo que hace sorprendente y único al Juego de la Vida es la capacidad de generar estructuras complejas a partir de un estado inicial aleatorio.

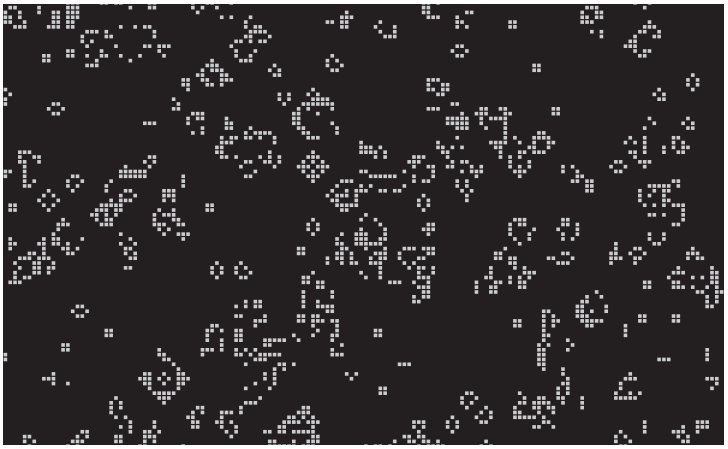

Figura 3.2: Ejemplo un universo del Juego de la Vida

Es decir, de un estado aparentemente caótico, después de un cierto tiempo, el universo tiende a formar siempre los mismos patrones. En concreto, hay 1517 patrones descubiertos documentados en la comunidad virtual [12], en la que se siguen haciendo nuevos hallazgos hasta día de hoy. En el siguiente apartado se muestra un resumen de la clasificación de patrones (*oscillators*, *still lifes*, *spaceships*), así como otras estructuras importantes que se pueden construir.

3.2. Clasificación de patrones

3.2.1. *Oscillators*

Los osciladores son aquellos que tras una serie de iteraciones, vuelven a tener la misma disposición de células en la misma posición inicial. Además, cuentan con tres parámetros:

- *Periodo*: el número de iteraciones que tiene que pasar para volver a la configuración inicial.
- *Stator*: todas aquellas células que permanecen vivas durante todas sus fases.
- *Rotor*: aquellas células del patrón que cambian de estado en alguna iteración.

El ejemplo más frecuente son los denominados *blinkers*. Como se observa en la figura 3.3, se trata de un patrón relativamente simple compuesto por 3 células vivas que cambian de una orientación vertical a uno horizontal en cada fase. Pertenece al grupo de osciladores de periodo 2, el más pequeño de esta clasificación.

Para entender mejor cómo funcionan estos patrones, de vuelta en la figura 3.3, resulta relativamente simple predecir su estado futuro aplicando las reglas:

- La célula superior en la primera fase tiene solo una vecina viva, al igual que la inferior, por lo que en en siguiente instante de tiempo morirán.
- La célula del medio tendrá siempre dos vecinas vivas, por lo que seguirá viva.
- La de la izquierda y derecha de ésta última, que se encuentran muertas, nacerán ya que cuentan justo con 3 vecinas vivas.

Cuando está en su configuración horizontal, se aplican exactamente las mismas reglas. De aquí se puede deducir que la célula central pertenecerá al *stator* del patrón, y las otras dos al *rotor*.

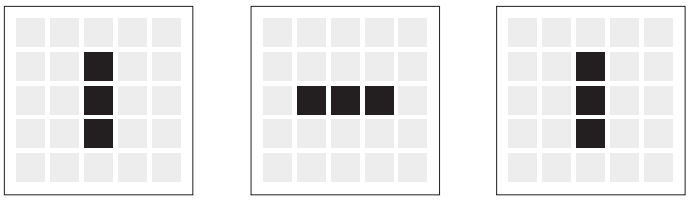

Figura 3.3: Ejecución de un *Blinker*, de izquierda a derecha.

Existen osciladores más complejos y de todos los periodos posibles. En la figura 3.4 se muestran más configuraciones de periodo 6.

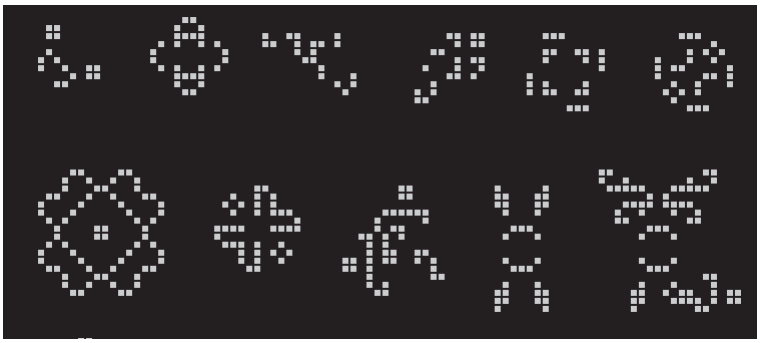

Figura 3.4: Ejemplos de osciladores de periodo 6.

3.2.2. *Still lifes*

Son conjuntos de células que forman estructuras estáticas, es decir, no cambian con el tiempo. Verdaderamente, pueden ser considerados como osciladores de periodo 1. Entre los comunes se encuentran los patrones de *beehives* (3.5) y *blocks* (3.6).

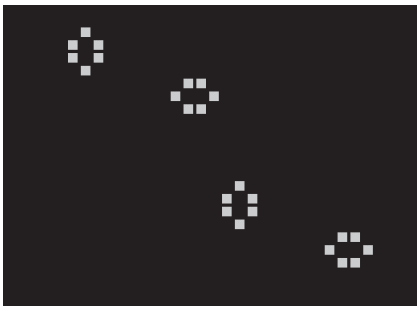

Figura 3.5: 4 patrones estáticos, *beehives*

3.2.3. *Spaceships*

Al igual que en el caso de los osciladores, este tipo de patrones cuenta con un periodo tras el que vuelven al estado inicial. No obstante, a diferencia de los primeros, al repetirse se

Figura 3.6: 4 patrones estáticos, *blocks*

encuentran en una posición diferente. Esta traslación permite que dichos patrones puedan moverse aparentemente a lo largo del universo. El ejemplo más conocido de esta clasificación son los *gliders* (figura 3.7). Además, a este tipo de patrones se le añade una característica más, la *velocidad*. Este parámetro se calcula al tomar la distancia (medida en nº de celdas) entre la disposición original y su trasladado, y dividirlo por su periodo. Se expresa en términos de la velocidad máxima posible c, es decir, moverse una célula por generación. Por ejemplo, en el caso del *glider* (figura 3.7) se observa que cada 4 iteraciones vuelve a su estado inicial, una célula por debajo en diagonal. Por lo tanto:

$$Velocidad_{glider} = \frac{1}{4}c$$

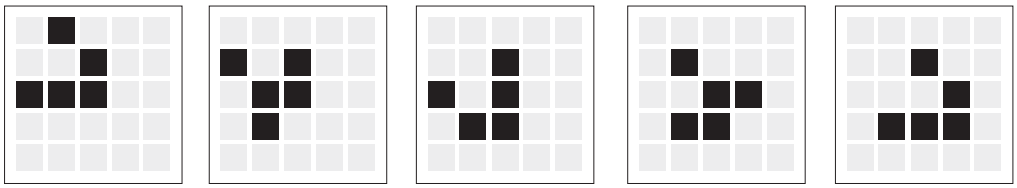

Figura 3.7: Ejemplo de ejecución de un Glider, de izquierda a derecha.

3.2.4. Configuraciones complejas

Como se mencionó en el apartado de autómatas celulares, Bill Gosper descubrió el primer patrón capaz de crecer infinitamente, llamado *Gosper Glider Gun*. En concreto, se trata de una configuración de 36 células que genera un *glider* cada 30 iteraciones (figuras 3.8 y 3.9).

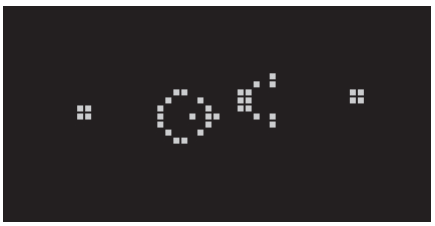

Figura 3.8: Configuración inicial de un *Gosper Glider Gun*.

Sin embargo, lo más notable de este tipo de patrón es la posibilidad de ser utilizado para imitar estructuras análogas a la computación. Haciendo una correspondencia con la

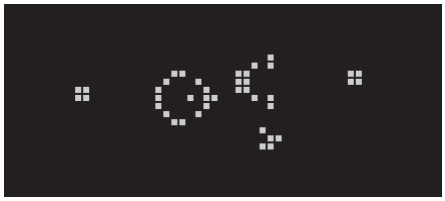

Figura 3.9: Configuración de un *Gosper Glider Gun* tras generar un *Glider*.

corriente eléctrica, se puede decir que la presencia de un flujo de *gliders* se puede considerar como 1, y su ausencia, como 0. A partir de esta comparación, la construcción de puertas lógicas queda determinada por la disposición de varias *Gosper Glider Gun*.

En las figuras 3.10 y 3.11 se representa la creación de una puerta NOT mediante diferentes *Gosper Glider Gun* [13]. Para entender mejor su funcionamiento, se ha divido la imagen en 4 partes (en las figuras se representan en color azul), siendo 1 la entrada de la puerta, y 4 su salida. Como es conocido, el comportamiento de una puerta NOT es devolver la entrada negada. Dicho esto:

- Cuando en 1 no hay un flujo de *gliders* (figura 3.10) el flujo de 2 corta al flujo de 3 por lo que los *gliders* provenientes de 4 siguen su camino hasta la salida.
- Por el contrario, cuando en 1 hay un flujo de *gliders* (figura 3.11), el flujo de 2 se ve afectado por lo que no es capaz de parar el proveniente de 3. En consecuencia, éste último elimina los *gliders* que nacen en 4, resultando en una ausencia de flujo en la salida.

El resto de puertas lógicas se puede construir de la misma manera. Una vez se cuenta con todas las configuraciones y un universo lo suficientemente grande, las posibilidades son infinitas. Desde la construcción de un sumador de 4 bits, hasta un ordenador completo. En definitiva, el Juego de la Vida es, realmente, una Máquina de Turing completa capaz de simular cualquier programa. Todo a partir de 4 sencillas reglas.

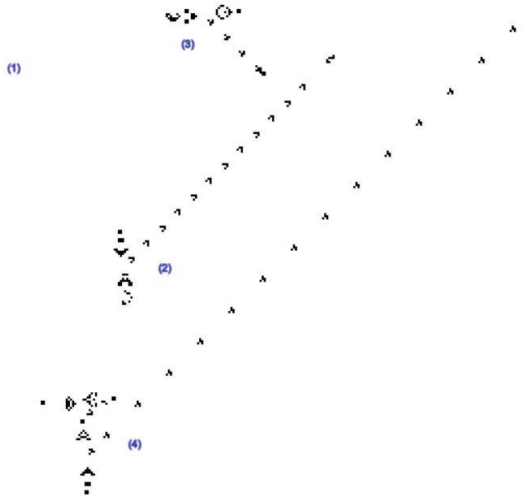

Figura 3.10: Puerta lógica NOT con entrada 0, salida 1.

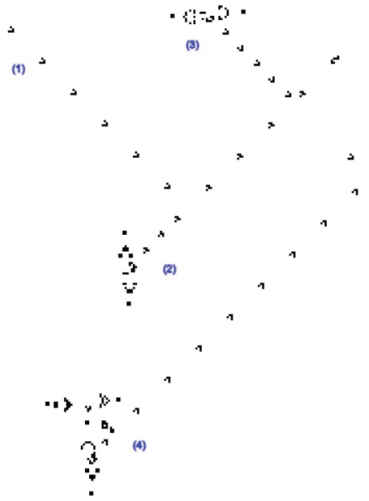

Figura 3.11: Puerta lógica NOT con entrada 1, salida 0

3.3. Definición de autómata celular abstracto

La definición de un autómata celular abstracto tiene que recoger las características del Juego de la Vida. En definitiva, los ingredientes principales son:

- un conjunto de celdas;
- los estados posibles de las celdas (vivo o muerto);
- una noción de adyacencia o vecindad;
- y una regla que determine el nuevo estado de cada celda en función de sus vecinas.

Además, hay unos principio de "homogeneidad" sobre estos cuatro objetos:

- Para cada celda, la noción de vecindad ha de ser la misma;
- y la regla que se aplica es la misma en todas las celdas.

Estas reglas implican que las celdas son indistinguibles entre sí, no hay ninguna que destaque sobre las demás. Esta homogeneidad en el conjunto de celdas se consigue con el conjunto de traslaciones, que me lleva de una celda a otra. Para cada $\vec{v} \in \mathbb{R}^2$, denotamos por $T_{\vec{v}}\colon \mathbb{R}^2 \to \mathbb{R}^2$ a la aplicación afín $T_{\vec{v}}(x) = x + \vec{v}$. Por tanto, el conjunto de todas las traslaciones por vectores cuyas coordenadas son enteras,

$$\{T_{\vec{v}}\colon \mathbb{R}^2 \to \mathbb{R}^2 \,:\, \vec{v} = (a, b) \text{ con } a, b \in \mathbb{Z}\},$$

permite llegar de una celda a otra. Lo interesante es que a este conjunto se le puede dotar de una estructura de grupo con la composición de aplicaciones afines: en este caso, $T_{\vec{v}} \circ T_{\vec{w}} = T_{\vec{v}+\vec{w}}$. Éste es el *grupo de las traslaciones*, y es *isomorfo* a $(\mathbb{Z} \times \mathbb{Z}, +)$.

Consideremos ahora la celda que contiene al origen de coordenadas $O = (0, 0)$: a través de la traslación $T_{\vec{v}}$ con $\vec{v} = (a, b) \in \mathbb{Z} \times \mathbb{Z}$ podemos ir de la celda del punto $(0, 0)$ a la del punto (a, b). Por eso, podemos identificar cada celda con un elemento del grupo de traslaciones: a

la celda con el punto $(a, b) \in \mathbb{Z}$ lo identificamos con la traslación $T_{\vec{v}}$ con $\vec{v} = (a, b)$, que es la que lleva la celda del $(0, 0)$ a la del (a, b). Este cambio de vista es interesante, ya que hemos sustituido el conjunto de celdas por un grupo, que es un conjunto con una estructura, y dicho grupo lleva consigo la idea de *homogeneidad* del espacio. Por ello, consideraremos como "celdas" a los elementos de un grupo G, y el propio G se convertiría en el espacio de celdas. Señalamos que dentro del grupo siempre hay un elemento destacado, que es el elemento identidad $e \in G$. Observa además que multiplicando por $g \in G$ por la izquierda, de e pasamos a g: $g \cdot e = g$.

Por otro lado tenemos además que las celdas tienen dos estados: viva o muerta, esto es, hay un conjunto de estados posibles $A = \{\text{vivo}, \text{muerto}\}$. En nuestra formulación más general, podemos considerar un conjunto de estados puede ser cualquier otro conjunto A, que se llamará *alfabeto*, *letras*, *estados*, *símbolos* o *colores*.

Una configuración del espacio en el juego de la vida de Conway es una asignación que a cada celda le da un valor "vivo" o "muerto", es decir, un valor del conjunto $A = \{\text{vivo}, \text{muerto}\}$. Dicho de otro modo, una configuración es una aplicación $x\colon G \to A$. En matemáticas al conjunto de aplicaciones de G a A se le denota por A^G. En nuestro contexto este conjunto A^G se corresponde con el conjunto de todas configuraciones posibles.

Las traslaciones permiten llevar una configuración de un punto a otro en el juego de la vida de Conway. Es decir, el grupo actúa sobre las configuraciones. En un marco más general, diremos que el grupo G actúa sobre A^G del siguiente modo: para cada configuración $x\colon G \to A$ y $g \in G$ un elemento del grupo, definimos $g^*x\colon G \to A$ como

$$(g^*x)(h) = x(g \cdot h).$$

De este modo, con la acción del grupo, podemos hacer que lo que ocurra en cualquier punto $g \in G$, llevarlo al punto $e \in G$:

$$\left((g^{-1})^*x\right)(g) = x(g^{-1} \cdot g) = x(e).$$

Dicho de otra forma: lo que ocurre en el punto $g \in G$ para la configuración de x, es lo mismo que lo que ocurre en $e \in G$ para la configuración g^*x. Así, sólo tendremos que tener en cuenta lo que ocurre en e.

Por otro lado, la evolución de las configuraciones se puede describir perfectamente como una función $\tau\colon A^G \to A^G$. Es decir, si $x\colon G \to A$ es una configuración, la siguiente configuración será $\tau(x)$. Y podemos aplicar τ sucesivamente para obtener las siguientes configuraciones. Así, tenemos una sucesión de configuraciones $\{x_n\colon G \to A\}_{n\in\mathbb{N}}$ determinado por la ley recursiva

$$\begin{cases} x_0 = x, \\ x_{n+1} = \tau(x_n), \end{cases}$$

donde $x\colon G \to A$ es la configuración inicial con la que partimos. Esta función τ tiene que estar determinada por una regla que no dependa de la celda $g \in G$, sino que la regla que apliquemos para averiguar el siguiente estado es la misma para todos las celdas de G. Por eso, sólo definiremos la regla en la celda identidad $e \in G$.

Dada la configuración $x_n\colon G \to A$, el valor en $e \in G$ de la siguiente configuración $x_{n+1}\colon G \to A$ dependerá de la vecindad de e, es decir, de las celdas que consideramos

adyacentes a e. Denotaremos por $S \subseteq G$ a la vecindad de e: en el juego de la vida de Conway, las celdas adyacentes son la propia celda y las 8 que la rodean. La regla que determinará el siguiente valor de x_{n+1} en e sólo tendrá en cuenta lo que ocurre en la vecindad S, por tanto la regla μ tomará la configuración que ocurre en S, $x_S \colon S \to A$, y dará el estado que le corresponderá en la siguiente configuración. Por tanto, una regla será una aplicación

$$\mu \colon A^S \to A.$$

Así,

$$x_{n+1}(e) = \mu(x_n|_S),$$

donde $x_n|_S \colon S \to A$ denota la restricción de la configuración $x_n \colon G \to A$ al conjunto $S \subseteq G$. Un ejemplo de regla es el juego de la vida de Conway que se define como

$$\mu(x) = \begin{cases} 1 & \text{si } \sum_{s \in S} x(s) = 3 \text{ y } x(\vec{0}) = 0, \\ 1 & \text{si } \sum_{s \in S} x(s) = 2, 3 \text{ y } x(\vec{0}) = 1, \\ 0 & \text{en otro caso.} \end{cases}$$

Usando la acción del grupo, llevamos los valores de la celda $g \in G$ (así como su entorno $gS = \{g \cdot s \in G \, : \, s \in S\}$) a la celda $e \in G$, y mediante μ determinamos su valor en el siguiente tiempo:

$$x_{n+1}(g) = \mu(((g^{-1})^* x_n)|_S).$$

3.4. Implementación

Para la implementación se ha decidido usar Python por dos razones. En primer lugar, es un lenguaje que ha ganado mucha popularidad en los últimos años por lo que cuenta con una gran cantidad de librerías y herramientas que facilitan diferentes tareas como, por ejemplo, ordenar vectores según un parámetro o gestionar una base de datos de forma sencilla. Por otro lado, tiene una sintaxis muy fácil de entender y directa lo que resulta de gran ayuda cuando hay que hacer un algoritmo complejo y no importa tanto la velocidad de ejecución.

En este caso, la librería principal es OpenGL, una interfaz que permite la creación de programas que produzcan gráficos en 2D o 3D. A día de hoy, aplicaciones como *Blender* o *CAD* y juegos como *Doom 2016* o *Minecraft* usan OpenGL para generar y optimizar los gráficos por pantalla.

Para representar una regla en concreto se usará un diccionario formado por una tupla como clave y un estado (0 ó 1, viva o muerta) como valor. La lista tendrá 9 elementos, representando un vecindario de izquierda a derecha y de arriba a abajo. El valor de cada clave será el estado futuro de la célula central teniendo en cuenta sus células vecinas. Cada diccionario contendrá entonces, $2^9 = 512$ pares de clave/valor. Se ha escogido un diccionario ya que cuentan con una complejidad $O(1)$ para acceder a un elemento [14]. Un ejemplo de un elemento de una regla podría ser:

```
1     rule = {
2         (0, 0, 1, 1, 0, 0, 0, 1, 1) : 1,
3         ...
4     }
```

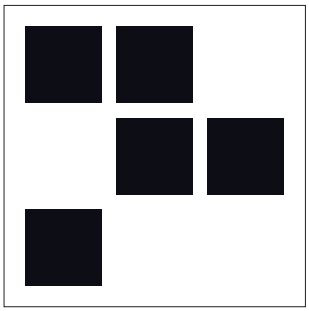

Figura 3.12: Contexto de la regla representada en código

En dicho caso, el contexto de la figura resultará en que la célula central obtendrá el estado de 1 en el siguiente instante, es decir, pasará de viva a muerta. La figura 3.12 muestra el estado del contexto mencionado.

3.4.1. Separación en módulos

La idea más sencilla para realizar el programa sería tener en un mismo fichero la lógica de evolución del universo con los gráficos que se muestran por pantalla ya que realmente no tienen mucha complejidad. No obstante, como se explicará más adelante en el capítulo de "Algoritmo Genético", el algoritmo necesitará hacer simulaciones de universos con diferentes reglas. Debido a esto, para evitar tener que repetir código se ha decidido separar la lógica del universo con la parte responsable de pintar las células por pantalla.

Por lo tanto, la implementación vendrá dada por dos módulos principales: `game.py` y `universe.py`. El primero será el que use las funciones de librería de OpenGL para crear la ventana y representar gráficamente el estado del universo; mientras que el segundo tendrá el código responsable de crear el siguiente estado fijada una regla y un universo actual. Este último módulo será utilizado por el algoritmo para realizar las simulaciones pertinentes.

3.4.2. universe.py

Este fichero será el encargado de contener la lógica del universo y cómo será el siguiente estado. Para ello, está formado por dos funciones:

- **nextState(rule, universe)**
  Recibe como argumento una regla y una matriz con el estado actual del universo, y devuelve el siguiente estado. Comprueba el estado del vecindario buscándolo en el diccionario que contiene la regla, y creará el próximo estado del universo.
- **_getContext(x, y, universe)**
  Toma la posición de una célula del universo y devuelve una lista con las 9 celdas que forman el vecindario. Será llamada desde **nextState** para cada una de células.

3.4.3. game.py

El código principal del programa se encontrará en este fichero, por lo que será el que hay que ejecutar. A grandes rasgos, el único objetivo que tiene es la de inicializar el universo, una matriz de $M \times N$ elementos (células), y en cada iteración pintar por pantalla una

representación del estado. En una primera configuración, los parámetros iniciales como la dimensión de la pantalla o el número de células en cada eje se controlarán directamente desde el código. A continuación, se muestra un fragmento de código con todos los parámetros iniciales:

```
1     BLACK = [0.0, 0.0, 0.0]
2     WHITE = [1.0, 1.0, 1.0]
3     GRAY = [0.15, 0.15, 0.15]
4
5     winColor = BLACK
6     gridColor = GRAY
7     cellColor = WHITE
8
9     # Window dimensions
10    dimXwindow = 1000
11    dimYwindow = 1000
12
13    # Universe dimensions
14    dimXgrid = 100
15    dimYgrid = 100
16    cellSize = dimXwindow / dimXgrid
17
18    # First state of the universe
19    universe = [[ 1  for i in range(dimXgrid)] for j in range(dimYgrid)]
20
21    # Execution speed
22    speed = 0.1
23    # Boolean that controls execution stop
24    stop = True
25    # Controls wether the grid is painted or not
26    paintGrid = True
```

La configuración anterior creará una pantalla de 1000 píxeles de alto y ancho, con una matriz de 100×100 células. Los estados de todas las células se guardarán en una matriz `universe`, pudiendo tomar los valores de: 0 (viva) ó 1 (muerta). Cada estado se mostrará cada 0,1 segundos siempre y cuando `stop` tenga valor `False`. Como se indica en el comentario, la variable `paintGrid` controla si se pinta una rejilla por pantalla, pudiendo observar los límites de cada célula.

En OpenGL no hay una relación directa entre las coordenadas de la pantalla y las coordenadas del dibujo. En otras palabras, si se quiere pintar un punto en el píxel con posición $(200, 200)$ en la pantalla, en el dibujo podrá tener las coordenadas $(1.5, 1.5)$, por ejemplo. Teniendo en cuenta las variables de `dimXwindow` y `dimYwindow`, sabemos que la pantalla irá desde 0 a `dimXwindow` píxeles de ancho y de 0 a `dimYwindow` píxeles de alto. Además, se sabe que las coordenadas que se verán del dibujo se indican con la siguiente función:

```
gluOrtho2D(left, right, bottom, top)
```

Se verán las posiciones desde `left` a `right`, y de `bottom` a `top`. De este modo, se podría pensar que la mejor manera de hacer la correspondencia sería de la siguiente forma:

```
gluOrtho2D(0, dimXwindow, 0, dimYwindow)
```

Al ejecutar la función con esos parámetros, la esquina inferior izquierda contendrá la posición $(0, 0)$ del dibujo y la superior derecha, la (`dimXwindow`, `dimYwindow`). Sin embargo, en OpenGL la posición $(0, 0)$ en píxeles se encuentra en la esquina superior izquierda, es decir, los píxeles que se encuentran más arriba en la pantalla tienen un valor más pequeño

que los de abajo. Al estar alteradas las coordenadas en el eje y, una solución sencilla es "invertir" el dibujo para que cuadre con las posiciones de la pantalla:

```
gluOrtho2D(0, dimXwindow, -dimYwindow, 0)
```

En consecuencia, la posición de la pantalla $(0, 0)$ será correspondida con la posición $(0, 0)$ del dibujo. Esto favorece las situaciones en las que, por ejemplo, se quiera pintar un cuadrado en unas coordenadas específicas al hacer click en la pantalla.

La ejecución del programa viene dada por un bucle creado por OpenGL al que hay que pasarle qué función queremos que se ejecute continuamente, a través de `glutDisplayFunc()`. Dicha función es la siguiente:

```
1     def main():
2         global universe
3         global stop
4         global rule
5
6         glClear(GL_COLOR_BUFFER_BIT)
7
8         if paintGrid:
9             grid()
10
11        drawAliveCells()
12
13        if not stop:
14            time.sleep(speed)
15            universe = u.nextState(rule, universe)
16            glutPostRedisplay()
```

En cada iteración del bucle, se limpiarán los elementos dibujados en la pantalla. Seguidamente, si se quiere pintar la rejilla se llamará a la función correspondiente `grid()`. Esta recorrerá de izquierda a derecha y de arriba a abajo, trazando rectas separadas entre sí según lo que mide el ancho de las células. Después, se realizará una llamada a `drawAliveCells()` que recorre la matriz del universo y pinta un cuadrado en las coordenadas con estado 0. Finalmente, si la ejecución no está parada, el proceso duerme los segundos correspondientes de `speed` y se hace una llamada al fichero `universe.py` para saber cuál será el siguiente estado del universo y qué regla debe usar. La llamada a `glutPostRedisplay()` es necesaria para que se vuelva a llamar a `main` en la siguiente iteración.

Finalmente, el código principal que incluye la configuración de OpenGL será:

```
1
2     # Window setup
3     def init():
4         global winColor
5         glClearColor(winColor[0], winColor[1], winColor[2], 1)
6         glMatrixMode(GL_PROJECTION)
7         gluOrtho2D( 0, dimXwindow, -dimYwindow, 0 )
8
9
10    glutInit(sys.argv)
11    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
12    glutInitWindowPosition(0,0)
13    glutInitWindowSize(dimXwindow,dimYwindow)
14    glutCreateWindow(b'Game of Life')
15
16    init()
17
```

```
18      glutDisplayFunc(main)
19      glutMouseFunc(mouseHandler)
20      glutKeyboardFunc(keyboard)
21      glutSpecialFunc(specialKeyboard)
22      glutMainLoop()
```

Las funciones `glutMouseFunc()`, `glutKeyboardFunc()` y `glutSpecialFunc()` son las que implementan los eventos de teclados y ratón, y, por lo tanto, los controles del programa.

3.4.4. Controles del programa

Para poder controlar el programa en tiempo de ejecución, se han implementado una serie de controles por teclado y ratón.

- **Ratón**
  - **Click izquierdo**: mediante el click sobre una célula, ésta invertirá de estado, de viva a muerta y viceversa.
- **Teclado**
  - **Tecla "C"**: todas las células del universo pasan al estado 1, es decir, muertas.
  - **Tecla "R"**: el universo pasa a tomar un estado aleatorio. Una de cada seis células vivirá.
  - **Tecla "S"**: similar al comportamiento anterior, pero en este caso sólo podrán nacer células en el espacio central de 40×40 células.
  - **Espacio**: la ejecución del programa se detiene. Se puede seguir "reviviendo" células mediante el ratón. Si estaba parado, se vuelve a reanudar.
  - **Tecla "M"**: cambia el color del autómata celular. Por defecto, las células se pintan de blanco en un fondo negro. Al pulsarse, se invierten los colores.
  - **Flecha hacia arriba**: aumenta la velocidad de ejecución.
  - **Flecha hacia abajo**: disminuye la velocidad de ejecución.
  - **Flecha derecha**: si la ejecución está parada, avanza al estado siguiente del universo.

3.4.5. Gestión de los datos

En el estado actual de desarrollo del programa no es estrictamente necesario tener una base de datos ya que solo se necesita guardar la regla del Juego de la Vida. No obstante, de cara al algoritmo evolutivo y el futuro desarrollo del proyecto, estaría bien tener la capacidad de guardar estados iniciales de un universo, patrones, reglas, etc. Para ello, se ha decidido desplegar un contenedor de *docker*, que instale y arranque una base de datos de *mongoDB*. La elección de *mongoDB* en vez una *SQL* se debe a que los datos guardados no van a estar prácticamente relacionados entre sí. Además, Python cuenta con la librería `pymongo` que permite el control de los datos de una manera sencilla.

Por esta razón, antes de ejecutar el programa por primera vez, hay que levantar el contenedor. Para ello se cuenta con la herramienta de *docker-compose*. El comando a ejecutar será:

```
docker-compose -f docker-compose.yml up -d
```

El fichero `docker-compose.yml` se encuentra en el directorio raíz del programa y cuenta con el siguiente contenido:

```
version: '3'
services:
  mongo:
    image: mongo
    restart: always
    container_name: mongo
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: root
    networks:
      - app-tier
    volumes:
      - ./data/mongodata:/data/db
    ports:
      - 27017:27017

networks:
  app-tier:
    driver: bridge
```

Si se quiere detener el contenedor:

```
docker-compose -f docker-compose.yml down
```

3.4.6. Ejemplos de ejecución

Una vez levantado el contenedor, simplemente hay que ejecutar el comando `python game.py`. A continuación, se muestra un ejemplo de ejecución con una pantalla de 1500×1000 píxeles, y de 150 células de ancho y 100 de alto.

Como se puede observar en la Figura 3.15, el programa genera los patrones típicos del Juego de la Vida como un *glider* abajo a la derecha o los patrones estáticos que forman un cuadrado de cuatro células.

Figura 3.13: Resultado de ejecutar un juego de pantalla de 1500×1000 píxeles

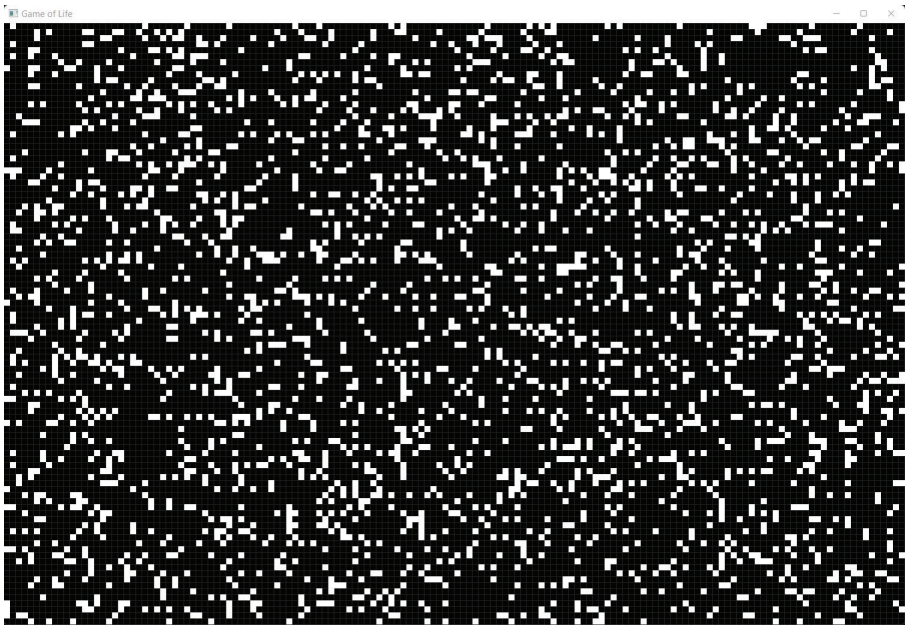

Figura 3.14: Estado aleatorio del universo.

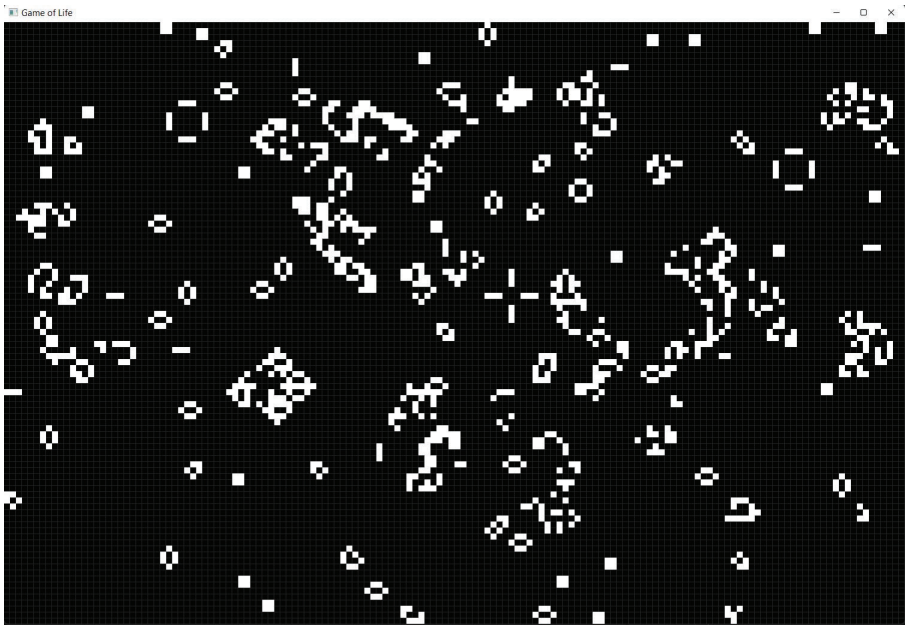

Figura 3.15: Estado del universo anterior tras 100 iteraciones.

Capítulo 4

Algoritmo Genético

En este capítulo vamos a detallar el algoritmo evolutivo que usaremos para hallar nuevas reglas que tengan la capacidad de formar patrones dinámicos. Para ello implementaremos una versión del algoritmo evolutivo desarrollado por los los autores Sapin, Bailleux y Chabrier [1].

4.1. Introducción a los algoritmos evolutivos

Como es bien sabido en informática, a la secuencia ordenada de pasos para lograr un objetivo se le conoce como algoritmo. Específicamente, los algoritmos evolutivos (EA, de *Evolutionary Algorithms*) intentan imitar a la naturaleza para encontrar los mejores resultados posibles que cumplan unas condiciones determinadas. Debido a esto, este tipo de algoritmos son realmente métodos de búsqueda que, mediante una serie de condiciones y modificaciones aleatorias, son usados para encontrar un resultado optimizado. Durante este capítulo se va a trabajar con un tipo particular de *EA*, los algoritmos genéticos.

En concreto, los algoritmos genéticos o *GA*, acrónimo de *Genetic Algorithms*, son aquellos que modelan la evolución y adaptación de los seres vivos. Partiendo de una población inicial de individuos o fenotipos (cada uno con sus características o genes), se pretende generar nuevos individuos que cuenten con las mejores características de la población anterior, así como nuevas mutaciones aleatorias. Dichas mutaciones aportan la exploración hacia nuevos caminos que puedan llevar a ejemplares que estén mejor preparados. En la figura 4.1 se muestra el esquema básico de todo *GA*.

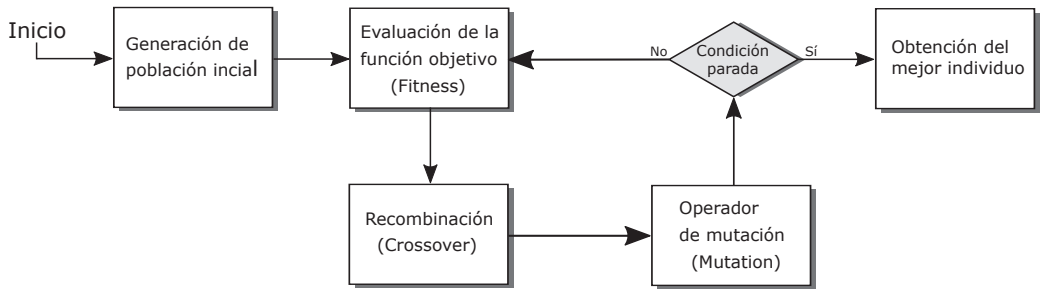

Figura 4.1: Esquema de un Algoritmo Genético.

En primer lugar y como se ha mencionado anteriormente, se selecciona una población inicial de muestras. A continuación, a cada miembro se le aplica una función objetivo que evaluará cómo de óptimo es. Si se trata de maximización, cuanto mayor sea el resultado, mejor se adapta el individuo a las condiciones propuestas. Una vez evaluada toda la población se selecciona a los mejores resultados, es decir, a los mejor adaptados. El resto son descartados.

Seguidamente se lleva a cabo una recombinación equivalente a la reproducción de los organismos vivos. Usualmente de los mejor adaptados (aquellos que no han sido descartados) se les empareja de manera aleatoria y por cada par se obtiene un individuo nuevo compuesto por una mezcla de las características de ambos padres. Por último, la población sufre cambios aleatorios de sus características permitiendo así ampliar el espacio de búsqueda. Esta secuencia de pasos puede repetirse hasta generar un individuo cuyo resultado sobrepase un límite preestablecido o hasta completar un número concreto de iteraciones. Si está bien implementado, por lo general cada generación tiende a obtener mejores resultados que la anterior.

4.2. Objetivos y diseño del algoritmo.

El objetivo principal del algoritmo será encontrar nuevas reglas distintas al Juego de la Vida que favorezcan la aparición del patrón de tipo *spaceships* (ver apartado 3.2.3) en un autómata celular de celdas cuadradas y dos estados (vivo o muerto). Como se vio en el capítulo anterior, dichos patrones son el primer paso para proporcionar capacidad de cómputo a un autómata celular.

Antes de explicar el diseño e implementación, es importante entender cómo se representa una regla. Anteriormente se ha explicado el término de vecindario en el que la célula central está rodeada por ocho células vecinas, por lo que forman conjuntos de nueve individuos. Luego el número total de vecindarios posibles es $2^9 = 512$. En la implementación del Juego de la Vida (apartado 3.4) se mostró la construcción de un diccionario para guardar el estado que obtendrá la célula central del vecindario en el siguiente instante de tiempo. En este caso, para facilitar el manejo de datos en la ejecución del algoritmo, se tiene que cada regla estará representada por una lista de 512 elementos en el que cada posición representará un vecindario en específico. Cada elemento de la lista será un 0 ó 1, representando el valor que tendrá la célula central en el siguiente estado.

En la figura 4.2 se muestra un ejemplo para entender mejor cómo funciona. De arriba a abajo, se representan las seis primeras posiciones de la lista, a qué vecindario representa cada posición, y el próximo estado de la celda central para esa regla en concreto. Cada una de ellas tendrá sus propios elementos, pero todas las reglas comparten la posición que representa cada vecindario.

Por otra parte, otro de los términos que se debe entender antes de empezar a explicar el funcionamiento del algoritmo es el *Espacio de Bays*. Formalmente hablando, dicho espacio es el conjunto de reglas que se pueden expresar de la forma E_bE_h/F_bF_h. Dicho de otro modo, una célula viva seguirá viva mientras tenga entre E_b y E_h vecinas vivas; y una célula muerta revivirá si tiene entre F_b y F_h vecinas vivas. Por ejemplo, el Juego de la Vida podría ser expresado como 23/33.

Dicho esto, se puede obtener que el espacio total de reglas (incluyendo las del Espacio de

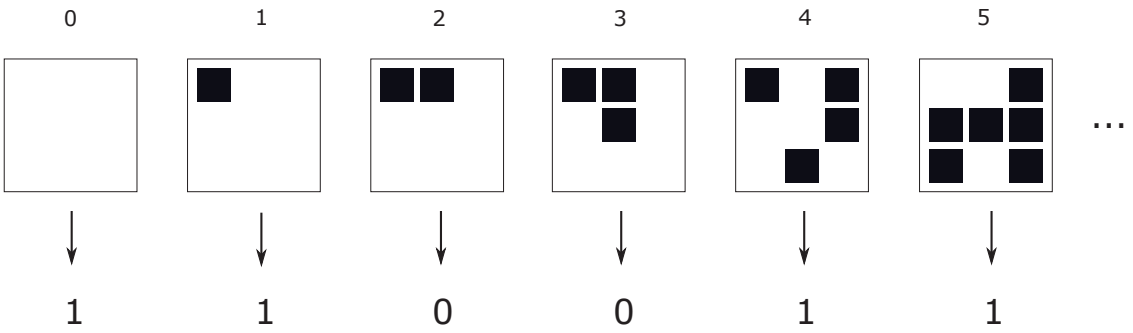

Figura 4.2: Ejemplo de regla que comienza por los elementos 110011

Bays) será de 2^{512} combinaciones posibles. Esto se puede observar mejor en la figura 4.3 donde se muestra cada uno de los espacios y la evolución que llevará a cabo el algoritmo. En definitiva, la regla resultante deberá ser una configuración que genere estructuras capaces de moverse a través del universo e, incluso, con la capacidad de autorreplicación.

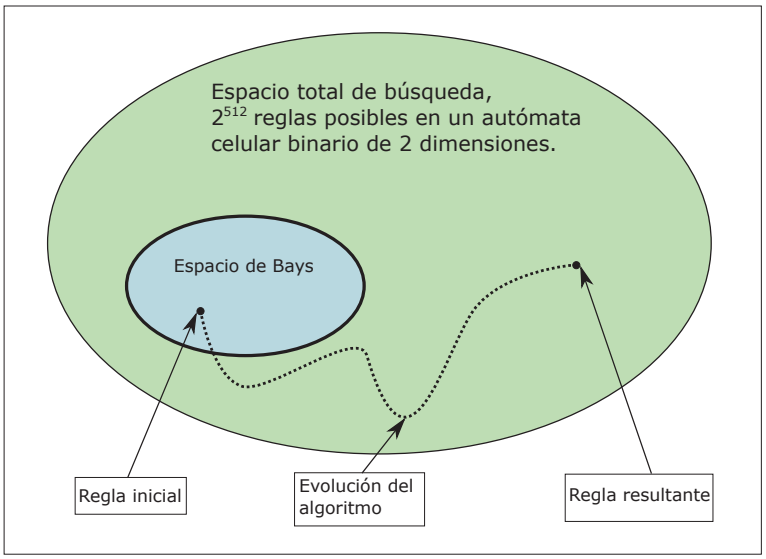

Figura 4.3: Diagrama del espacio total de búsqueda.

A continuación, se muestra el diseño del algoritmo por cada paso y operador (inicialización, *fitness*, *selection*, *crossover* y *mutation*).

4.2.1. Población inicial

El algoritmo partirá de una población inicial formada por 50 reglas usando el formato descrito en el apartado anterior. La composición de reglas será la siguiente:

- Diez copias iguales de una regla R perteneciente al Espacio de Bays.
- Diez variantes formadas a partir de una mutación de R.
- Diez variantes formadas a partir de dos mutaciones de R.
- Diez variantes formadas a partir de tres mutaciones de R.

- Diez variantes formadas a partir de cuatro mutaciones de R.

4.2.2. Función objetivo (*Fitness*)

A cada regla de la población se le debe asignar una puntuación que indique cómo de óptima es para poder seleccionar más tarde a los mejores individuos. A dicho valor se le denomina *fitness* y se consigue mediante una función objetivo idéntica para toda la población. Para ver qué parámetros se deben tener en cuenta en la función es importante realizar un estudio sobre las condiciones necesarias que se cumplen en el Juego de la Vida y que le permiten generar los patrones deseados. Podemos destacar las siguientes conclusiones:

- Al punto imaginario resultante de calcular la media de las posiciones de todas células vivas se le denomina *centro de gravedad*. Dicho punto se suele mantener en el centro del universo a lo largo de las iteraciones. En otras palabras, el autómata cambia de forma, pero el conjunto de células vivas no tiende a ninguna dirección en concreto.
- El número total de células vivas en cada iteración se mantiene estable. Con el tiempo la población va aumentando o disminuyendo, pero siempre en torno a un valor en concreto. Esto se debe a la aparición de patrones estables y de *gliders* y otras *spaceships* que, aunque se muevan, repiten las mismas estructuras al cabo de un tiempo.
- Para estudiar cómo crece el conjunto de células vivas se pueden observar los límites de la población. En otras palabras, podemos estudiar su crecimiento mediante un rectángulo imaginario que se ocupe de mantener en su interior todas las células vivas en cada iteración. Si aparece un *glider*, su movimiento en una dirección en concreto causará que el rectángulo crezca indefinidamente. Por el contrario, en una regla que no permita la aparición de patrones dinámicos el área del rectángulo se mantendrá estable o incluso disminuirá.

Una vez realizado el estudio, se debe diseñar cómo el algoritmo va a obtener la puntuación de cada regla. Para ello, por cada miembro de la población y por cada iteración, se ejecutará una simulación de un universo con la regla escogida. La simulación constará de una rejilla de 200×200 células, en el que se creará una "sopa primordial" en el centro. Esto será un cuadrado centrado de 40×40 en el que el estado inicial de las células interiores será aleatorio. El resto de células permanecerán muertas.

El primer parámetro que se quiere obtener será el que controle el crecimiento del cuadrado central. En cada iteración, se comprobará que todas las células vivas estén dentro del cuadrado. Si se detecta que el cuadrado es lo suficientemente grande, encogerá hasta que una o varias celdas vivas se encuentren justo en el límite. En caso de que existan células vivas fuera del cuadrado imaginario, éste crecerá hasta cubrirlas a todas. En resumen, se tendrá la siguiente expresión:

$$S_1 = \frac{N_1 - N_2 + L}{L} = \frac{N_1 - N_2}{L} + 1,$$

siendo N_1 el número total de cuántas veces crece el cuadrado central, N_2 de cuántas veces decrece, y L, el número de iteraciones totales de la simulación (en nuestro caso serán 200). Como se puede ver, S_1 será mayor si las células vivas tienden a desplazarse hacia los exteriores del cuadrado.

A continuación, para controlar la estabilidad de la población se debe tener en cuenta el número de células vivas actual y en la iteración anterior. Para ello, de una forma similar a la anterior, se tiene que:

$$S_2 = \frac{M_1 - M_2 + L}{L} = \frac{M_1 - M_2}{L} + 1,$$

siendo M_1 el número de veces que decrece la población de células vivas y M_2 el número de veces que crece.

Finalmente, para tener el cuenta el centro de gravedad se tiene el parámetro C_1 que se calcula:

$$C_1 = 1 + \frac{dg}{sz},$$

siendo dg la distancia euclídea entre el centro de gravedad de las células vivas y el centro físico del universo, y sz el tamaño de la simulación, en este caso 200. Juntando los parámetros anteriores se consigue la siguiente expresión:

$$F = \frac{S_1 \cdot S_2}{C_1}$$

El algoritmo intentará maximizar la función anterior, es decir, cuanto mejor puntúe una regla más óptima será.

En definitiva, con S_1 se garantiza que la regla resultante tienda a crecer hacia el exterior del cuadrado inicial de 40×40, con S_2 que la población no crezca ni decrezca demasiado y con C_1 que dicha población se mantenga cerca del centro del universo. De esta forma, obtendrán buenos resultados aquellas reglas que se mantengan con una población estable en el centro a lo largo de las iteraciones y que, de vez en cuando, generen un patrón dinámico que se extienda a los exteriores del universo.

4.2.3. Selección (*Selection*)

Para el operador de selección se ha decido usar una selección simple de las 20 mejores reglas. En otras palabras, después de calcular el *fitness* de toda la población, se ordena de mayor a menor y se cogen las 20 mejores. El resto de reglas serán descartadas.

4.2.4. Recombinación (*Crossover*)

A continuación, con las 20 reglas anteriores se hacen 10 parejas aleatorias. Estas parejas serán usadas para hacer una recombinación denominada *Single Point Crossover*. De cada pareja, se creará una nueva regla que será de nuevo una lista de 512 elementos usando los 256 primeros de un padre y los 256 siguientes del otro. De esta forma, se crearán 10 reglas nuevas.

4.2.5. Mutación (*Mutation*)

En este caso, una mutación consistirá en tomar una posición aleatoria de una regla e invertir su estado de 0 a 1 y viceversa. Como se explica en el artículo de Emmanuel Sapin, Olivier Bailleux y Jacqueline Chabrier [1], realizar sólo una mutación por cada regla y cada generación no es suficiente para ampliar el espacio de búsqueda. Por esta razón, por cada

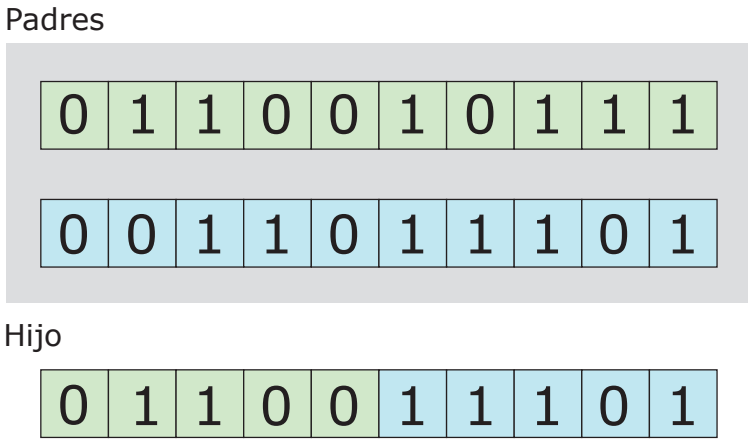

Figura 4.4: Ejemplo de recombinación *Single Point Crossover* de punto central.

pareja de reglas escogidas en el apartado anterior se calcula la distacia de Hammings entre ellas. Si la distancia es menor que 5, se aplicarán 5 mutaciones a ambas reglas. Si es mayor o igual, solo se realizará una.

La distancia de Hammings se calcula contando las diferencias que tienen dos secuencias binarias de números. Por ejemplo, la distancia entre 1101101 y 1111111 será de 2.

4.2.6. Condición de parada

El algoritmo parará su ejecución cuando se encuentre una regla cuyo *fitness* sea mayor a 1,75. Justo después, se insertará el resultado en la base de datos. No obstante, para evitar ejecuciones muy largas si la regla desde la que se parte no genera buenos resultados, al lanzar el programa se podrá indicar el número máximo de iteraciones. En ese caso, se guardará la lista con mejores resultados de la última iteración.

4.3. Ayudas para la implementación

Para ayudar a la implementación del algoritmo se han creado dos ficheros, `rule.py` y `dbManager.py`, que representan el concepto de una regla y llevan el manejo de la base de datos respectivamente. Como se va a hacer uso de ellos durante el resto del proyecto sería conveniente mostrar su funcionamiento con anterioridad.

4.3.1. Gestión de la base de datos

Tal y como se mencionó en el apartado del Juego de la Vida, se cuenta con una base de datos para guardar automática las reglas y contextos que se generen en el algoritmo. Para ello, como se trata de *MongoDB* se tiene que instalar y utilizar la librería de `pymongo`. A continuación se muestran los métodos realizados:

```
1 from pymongo import *
2 import json
3
4 def connect() -> MongoClient:
5     client = MongoClient("mongodb://root:root@localhost:27017")
```

```
6       db = client['gameDB']
7       return db
8
9   def insertRule(name, rule) -> None:
10      db = connect()
11
12      db["rules"].insert_one({
13          "name" : name,
14          "value" : rule
15      })
16
17  def insertContexts() -> None:
18      db = connect()
19
20      f = open('data/data.json')
21      data = json.load(f)
22      f.close()
23
24      db["contexts"].insert_one({
25          "contexts" : data["contexts"]
26      })
27
28  def getRuleByName(name) :
29      db = connect()
30      collection = db["rules"].find_one({"name" : name})
31      return collection["value"]
32
33  def getLifeRule():
34      return getRuleByName("life_rule")
35
36  def getContexts():
37      db = connect()
38      collection = db["contexts"].find_one()
39      return collection["contexts"]
```

Explicación por métodos:

- `connect()`: realiza la conexión a la base de datos que se ejecuta en el contenedor de *docker*.
- `insertRule(name, rule)`: guarda una regla con nombre `name` y asociada con la representación de su regla en forma de lista.
- `insertContexts()`: inserta los 512 contextos en la base de datos para tenerlos guardados de forma permanente. Sólo es necesario que se ejecute una vez ya que el orden de los contextos será el mismo para cada regla.
- `getRuleByName(name)`: dado un nombre de una regla devuelve la lista asociada a dicha regla. Si no existe, devuelve `None`.
- `getLifeRule()`: llama a la función anterior con el nombre `life_rule`. Sirve para la implementación del Juego de la Vida.
- `getContexts()`: devuelve los 512 vecindarios posibles.

4.3.2. Clase *Rule()*

Cada individuo de la población será una instancia de esta clase. En ella se contiene la lista que representa cada regla así como el diccionario necesario para representarse gráficamente.

De esta forma, se podrá usar el programa de `game.py` para que ejecute cualquier regla, cambiando el diccionario del Juego de la Vida por el deseado. El contenido es el siguiente:

```
from data import dbManager

class Rule():

    def __init__(self, rule):
        self.rule = rule
        self.contexts = dbManager.getContexts()
        self.ruleDictionary = self._generateDictionary()

    def _generateDictionary(self):
        dict = {}
        for (context, bit) in zip(self.contexts, self.rule):
            dict.update({
                tuple(context) : bit
            })
        return dict

    # returns the rule itself, the dictionary
    def getRule(self):
        return self.ruleDictionary

    # returns the state list of the rule
    def getRuleList(self):
        return self.rule

    # sets a new rule and creates a new dictionary
    def setRule(self, newRule):
        self.rule = newRule
        self.ruleDictionary = self._generateDictionary()
```

4.4. Implementación

En cuanto a la implementación del algoritmo en Python se utilizará un repositorio de GitHub para el control de versiones. Se puede acceder al mismo mediante el siguiente enlace: github.com/glopez42/game-of-life.git

Como se puede observar en la carpeta `/algorithm`, se ha creado un módulo con los siguientes ficheros:

- `crossover.py`: contiene los operadores de recombinación.
- `fitness.py`: contiene los operadores de evaluación.
- `mutation.py`: contiene los operadores de mutación.
- `selection.py`: contiene los operadores de selección.
- `simulation.py`: realiza las simulaciones pertinentes para poder evaluar a cada regla de la población. Para la lógica del universo usará el fichero `universe.py` mencionado en la implementacicón del Juego de la Vida.
- `main.py`: se encarga de la ejecución del algoritmo y realizar el bucle principal.
- `init.py`: inicializa la regla del Espacio de Bays desde la que se parte.

Las funciones de recombinación, evaluación, selección y mutación harán exactamente lo explicado en los apartados anteriores. No obstante, para separar la ejecución del algoritmo con el funcionamiento de cada uno de ellos se tendrá una clase abstracta por cada operador. De este modo, se pueden hacer cambios en su funcionamiento o programar diferentes clases con otros comportamientos sin afectar al algoritmo. A continuación, se muestra una explicación más exhaustiva de las partes del código con mayor relevancia.

4.4.1. crossover.py

```
from abc import ABC, abstractmethod
from rule import Rule

class Crossover(ABC):

    @abstractmethod
    def performCrossover(self):
        pass


class SinglePointCrossover(Crossover):

    def performCrossover(self, rule1: Rule, rule2: Rule):
        bitArray1 = rule1.getRuleList()
        bitArray2 = rule2.getRuleList()

        length = len(bitArray1)
        middle = int(length / 2)

        newBitArray = bitArray1[0:middle] + bitArray2[middle:length]

        return Rule(newBitArray)
```

Como se ha mencionado en el apartado anterior, el operador de `SinglePointCrossover()` implementa una clase abstracta denominada `Crossover`, por lo que tiene que tener el método de `performCorssover(R1, R2)`. Dicho método de la clase hija toma como parámetros las dos reglas padre, saca la mitad de la longitud de la lista y crea una regla nueva con la mitad de los elementos de cada padre.

4.4.2. fitness.py y simulation.py

Para evaluar a cada regla se tienen los ficheros de `fitness.py` y `simulation.py`. El primero recorrerá a cada individuo, lanzará una simulación y, una vez esta termine, calcula el resultado según los parámetros explicados en el apartado 4.2.2.

```
from abc import ABC, abstractmethod
from typing import Dict, List
from algorithm.simulation import Simulation
from rule import Rule
import timeit

class Fitness(ABC):

    @abstractmethod
    def fitnessFunction(self, population):
        pass


class FirstAproachFitness(Fitness):
```

```
15
16      def __init__(self):
17          self.iters = 200
18          self.width = 200
19          self.height = 200
20
21      # calculates fitness of each rule and returns a list containing all fitness
            results
22      def fitnessFunction(self, population: List[Rule]):
23
24          results = []
25          x = 1
26          for rule in population:
27              # runs a simulation of an universe during 200 iterations with that rule
28
29              print("\tSimulation {} out of 50 started...".format(x))
30              start = timeit.default_timer()
31              simulation = Simulation(self.iters, self.width, self.height, rule.
                    getRule())
32              parameters = simulation.runSimulation()
33              stop = timeit.default_timer()
34              print("\tSimulation ended in  " + str(stop - start) + " s")
35
36              fitness = self._calculate_fitness(parameters)
37              results.append((rule, fitness))
38
39              x += 1
40
41          return results
```

Al igual que el resto de operadores, se tiene una clase abstracta `Fitness` que deberá implementar. Por cada elemento de la población ejecutará una simulación con la variable `simulation` y su método `runSimulation()`. Al crear la clase `Simulation` se inicializará un universo inicial (tal y como se explica en el apartado 4.2.2) y el método `runSimulation()` se encargará de ejecutar las 200 iteraciones. A continuación se muestra su código:

```
1       def runSimulation(self):
2           for _ in range(self.iter):
3               self.universe = nextState(self.rule, self.universe)
4               self._check_limits()
5               self._check_population()
6
7           return self._return_parameters()
```

La función hará tantas iteraciones como se le indique en el atributo `iter` y, como se explicó en el apartado Separación en módulos (3.4.1), el algoritmo hará uso del fichero `universe.py` y su método `nextState()`. Por otra parte, por cada iteración se llamará a `_check_limits()` que ajusta los límites del cuadrado imaginario que debe contener a todas las células vivas (ver apartado 4.2.2). De la misma manera, `_check_population()` se encargará de ver si la población ha crecido o decrecido respecto a la última iteración. Al finalizar la simulación se devolverán los parámetros pertinentes en un diccionario creado en `_return_parameters()`. El código de las últimas funciones mencionadas no se muestra en la memoria por temas de simplificación, pero se puede consultar en el fichero `/algorithm/simulation.py` en el repositorio de GitHub.

Una vez terminada la simulación y de vuelta en el fichero `fitness.py`, se calcula el *fitness* usando la función privada `_calculate_fitness(parameters)` pasando como argumento el

diccionario con los parámetros de la simulación. Como se observa en el código, simplemente realiza las operaciones descritas anteriormente.

```
    def _calculate_fitness(self, parameters: Dict):
        n1 = parameters["N1"]
        n2 = parameters["N2"]
        m1 = parameters["M1"]
        m2 = parameters["M2"]
        L = parameters["L"]
        c1 = parameters["C1"]

        S1 = (n1 - n2 +  L) / L
        S2 = (m1 - m2 + L) / L
        f = (S1 * S2) / c1

        return f
```

4.4.3. mutation.py

En cuanto a las mutaciones, sencillamente se trata de obtener dos reglas, calcular la distancia de Hammings entre ellas, ver cuántas mutaciones hay que realizar y llevarlas a cabo:

```
from abc import ABC, abstractmethod
from rule import Rule
import random

class Mutation(ABC):

    @abstractmethod
    def performMutation(self):
        pass


class HammingDistanceMutation(Mutation):

    # performs the correspondent mutations in each rule
    def performMutation(self, rule1: Rule, rule2: Rule):
        bitArray1 = rule1.getRuleList()
        bitArray2 = rule2.getRuleList()

        distance = self._getHammingDistance(bitArray1, bitArray2)
        numberOfMutations = 1 if distance >= 5 else 5

        for _ in range(numberOfMutations):
            bitArray1 = self.mutate(bitArray1)
            bitArray2 = self.mutate(bitArray2)

        rule1.setRule(bitArray1)
        rule2.setRule(bitArray2)

        return rule1, rule2

    # Returns the hamming distance between 2 arrays, works like a XOR gate
    def _getHammingDistance(self, array1, array2):
        distance = 0
        for bit1,bit2 in zip(array1, array2):
            if bit1 != bit2:
                distance += 1
        return distance
```

A cada una de las dos, por cada número de mutaciones, se les aplica el método `mutate()`. Como se ha explicado antes, las mutaciones ocurren de forma aleatoria en cualquier elemento de la regla. El siguiente fragmento de código muestra la implementación de la función comentada:

```
    # Takes a random position of the array and reverse its state
    def mutate(self, bitArray):
        length = len(bitArray)
        randomPos = random.randint(0, length - 1)
        previousState = bitArray[randomPos]
        bitArray[randomPos] = 0 if previousState == 1 else 1
        return bitArray
```

4.4.4. selection.py

Para la selección, se toma la población (una lista de tuplas que contienen una regla y su *fitness*), se ordena de mayor a menor según su puntuación, se guardan las 20 mejores y se devuelven solo las listas. Adicionalmente, también se devuelve el mejor valor para que el fichero principal del algoritmo decida si parar o no la ejecución.

```
from abc import ABC, abstractmethod
from typing import Tuple
from typing import List

class Selection(ABC):

    @abstractmethod
    def makeSelection(self, population):
        pass

class SelectBest20(Selection):

    def makeSelection(self, population: List[Tuple]):
        sorted_by_fitness = sorted(population, key=lambda tup: tup[1], reverse=True
            )
        best20 = sorted_by_fitness[0:20]
        best = []
        bestFitness = best20[0][1]
        print("Best fitness: ", bestFitness)
        # removes the fitness value of each rule as it is not longer required
        for tuple in best20:
            best.append(tuple[0])
        return best, bestFitness
```

4.4.5. main.py y init.py

Finalmente, se tienen los ficheros `main.py` y `init.py` encargados de la ejecución del algoritmo y la inicialización de la regla inicial. A continuación se muestra el código correspondiente a `init.py`:

```
from data import dbManager
from rule import Rule

def aliveNeighbors(c):
    alive = 0
    # loops through the neighbors
    for i in range(9):
```

```
8           neighbor = c[i]
9           # if it's alive and is not the central cell
10          if neighbor == 0 and i != 4:
11              alive += 1
12      return alive
13
14  # Bays space rule: EbEh / FbFh
15  def getBaysSpaceRule(ruleID) -> Rule:
16          rule = []
17
18          Eb = ruleID[0]
19          Eh = ruleID[1]
20          Fb = ruleID[2]
21          Fh = ruleID[3]
22
23          contexts = dbManager.getContexts()
24          for c in contexts:
25                  # central cell
26                  cell = c[4]
27                  # if alive
28                  if cell == 0:
29                          alive = aliveNeighbors(c)
30                          if alive >= Eb and alive <= Eh:
31                                  rule.append(0)
32                          else:
33                                  rule.append(1)
34
35                  # dead
36                  else:
37                          alive = aliveNeighbors(c)
38                          if alive >= Fb and alive <= Fh:
39                                  rule.append(0)
40                          else:
41                                  rule.append(1)
42
43          return Rule(rule)
```

La función principal del código es `getBaysSpaceRule(rule)` que obtiene como parámetros una tupla (representando la regla del Espacio de Bays con formato E_bE_h/F_bF_h) y devuelve un objeto de tipo `Rule` con el diccionario de la regla pertinente. Para ello, por cada uno de los 512 contextos (almacenados en la base de datos) se cuentan los vecinos vivos y se añade un 0 ó 1 en la lista dependiendo de las características de la regla.

Lo único que haría falta para terminar el algoritmo sería juntar todos los operadores en el orden explicado anteriormente. Como recordatorio y siguiendo el diagrama de la figura 4.1, primero se inicializan los 50 individuos de la población inicial, se evalúa cada una de las reglas por separado, se escogen las 20 mejores y se descarta el resto. A continuación, se generan nuevos individuos mediante recombinaciones y mutaciones aleatorias y se vuelve a empezar. Si se detecta una regla con un *fitness* de más de 1,75 o se completa el número total de iteraciones, el algoritmo termina.

El fichero `main.py` contendrá la clase `Algorithm()`, a la que se le pasa como argumentos al constructor la regla inicial:

```
1 from algorithm.crossover import *
2 from algorithm.mutation import *
3 from algorithm.fitness import *
4 from algorithm.init import *
5 from algorithm.selection import *
6 import copy
```

```
import timeit


class Algorithm():

    def __init__(self, initialRule) -> None:
        self.iter = 0
        self.crossover = SinglePointCrossover()
        self.mutation = HammingDistanceMutation()
        self.fitness = FirstAproachFitness()
        self.selection = SelectBest20()
        self.population = []
        self.initialRule = initialRule
```

Además, como se observa, se crean las variables que contendrán los operadores necesarios para cada paso del algoritmo. Por otra parte, la clase contiene el método `_initPopulation()` encargado de generar los 50 individuos de la población inicial, tal y como se indica en la sección 4.2.1:

```
    def _initPopulation(self):
        # Initial rule from Bays space
        startRule = getBaysSpaceRule(self.initialRule)

        # First 10 rules without mutations
        for _ in range(10):
            rule = copy.deepcopy(startRule)
            self.population.append(rule)

        # Performs 10 single mutations
        for _ in range(10):
            startList = copy.deepcopy(startRule.getRuleList())
            bitArray = self.mutation.mutate(startList)
            self.population.append(Rule(bitArray))

        # Performs 10 double mutations
        for _ in range(10):
            mutatedRule = copy.deepcopy(startRule)
            bitArray = mutatedRule.getRuleList()
            for _ in range(2):
                bitArray = self.mutation.mutate(bitArray)
            self.population.append(Rule(bitArray))

        # Performs 10 triple mutations
        for _ in range(10):
            mutatedRule = copy.deepcopy(startRule)
            bitArray = mutatedRule.getRuleList()
            for _ in range(3):
                bitArray = self.mutation.mutate(bitArray)
            self.population.append(Rule(bitArray))

        # Performs 10 cuadruple mutations
        for _ in range(10):
            mutatedRule = copy.deepcopy(startRule)
            bitArray = mutatedRule.getRuleList()
            for _ in range(4):
                bitArray = self.mutation.mutate(bitArray)
            self.population.append(Rule(bitArray))
```

Para terminar, se cuenta con el método principal, `run(iterations)`, que ejecuta el bucle principal tantas veces como indique el parámetro de entrada `iterations`.

```
    def run(self, iterations) -> Rule:
```

```
2
3           self.iter = iterations
4           # sets up the population
5           self._initPopulation()
6
7           for i in range(self.iter):
8               start = timeit.default_timer()
9               print("##### Iter " + str(i) + " started #####\n")
10
11              # fitness and selection
12              print("Fitness started...")
13              startAux = timeit.default_timer()
14              fitnessResults = self.fitness.fitnessFunction(self.population)
15              stopAux = timeit.default_timer()
16              print("Fitness ended in  " + str(stopAux - startAux) + " s")
17
18              print("Selection started...")
19              startAux = timeit.default_timer()
20              selectionResults, bestFitness = self.selection.makeSelection(
                    fitnessResults)
21              stopAux = timeit.default_timer()
22              print("Selection ended in  " +  str(stopAux - startAux) + " s")
23
24              # if the fitness is already quite good
25              if bestFitness >= 1.75:
26                  break
27
28              # crossover
29              print("Crossover started...")
30              startAux = timeit.default_timer()
31              shuffledList = copy.deepcopy(selectionResults)
32              random.shuffle(shuffledList)
33              crossoverResults = []
34              for i in range(10):
35                  rule1 = shuffledList[i]
36                  rule2 = shuffledList[i + 10]
37                  result = self.crossover.performCrossover(rule1, rule2)
38                  crossoverResults.append(result)
39              stopAux = timeit.default_timer()
40              print("Crossover ended in  " +  str(stopAux - startAux) + " s")
41
42              # mutation
43              print("Mutation started...")
44              startAux = timeit.default_timer()
45              mutationResults = []
46              for i in range(10):
47                  rule1 = shuffledList[i]
48                  rule2 = shuffledList[i + 10]
49                  result1, result2 = self.mutation.performMutation(rule1, rule2)
50                  mutationResults.append(result1)
51                  mutationResults.append(result2)
52              stopAux = timeit.default_timer()
53              print("Mutation ended in  " + str(stopAux - startAux) + " s")
54
55              # new population
56              self.population = []
57              self.population += selectionResults
58              self.population += crossoverResults
59              self.population += mutationResults
60
61              stop = timeit.default_timer()
62              print("\t---> Finished iteration in " + str(stop - start) + " s")
63
64
65          print("\n#### Algorithm execution finished ####")
66          print("\n\tBest fitness: " + str(bestFitness))
67
```

```
68          # retrieves the best rule
69          return selectionResults[0], bestFitness
```

Además, como se observa en el fragmento de código, el algoritmo irá imprimiendo por pantalla la duración de cada parte, facilitando de este modo el seguimiento de la ejecución.

4.5. Ejecución del algoritmo y representación gráfica de reglas

Para ejecutar el algoritmo lo único necesario sería crear un objeto de la clase `Algorithm` y realizar la llamada al método `run()`. No obstante, se aporta un programa de prueba con el que se pueden realizar ejecuciones usando comandos por consola. Dicho programa tiene el siguiente contenido:

```
1  from algorithm.main import Algorithm
2  from algorithm import init
3  from data import dbManager
4  from rule import Rule
5  import sys
6
7  if len(sys.argv) != 4:
8      print("ERROR: start.py usage: \n\npython start.py [initial_rule] [result_name]
           [iterations]")
9      exit(-1)
10
11 iter = int(sys.argv[3])
12 initialRuleName = sys.argv[1]
13 splitRule = initialRuleName.split("/")
14
15 Eb = int(splitRule[0][0])
16 Eh = int(splitRule[0][1])
17 Fb = int(splitRule[1][0])
18 Fh = int(splitRule[1][1])
19 initialRule = (Eb, Eh, Fb, Fh)
20
21 # Algorithm
22 rule, fitness = Algorithm(initialRule).run(iter)
23 dbManager.insertRule(sys.argv[2], rule.getRuleList())
24 print("\nRule saved on the database with name: " + sys.argv[2])
25
26 # Mutations
27 rule2 = dbManager.getRuleByName(initialRuleName)
28 cont = 0
29 for bit1, bit2 in zip(rule.getRuleList(), rule2):
30     if bit1 != bit2:
31         print("Mutacion en bit: ", cont)
32     cont += 1
```

Por lo tanto, para realizar una llamada por comandos:

```
python start.py [initial_rule] [result_name] [iterations]
```

siendo:

- `initial_rule` la regla desde la que se parte, con formato E_bE_h/F_bF_h.
- `result_name` el nombre con el que se guardará en la base de datos
- `iterations` el número de iteraciones totales que se ejecutarán, como máximo.

Del mismo modo, para representar gráficamente el Juego de la Vida o cualquier otra regla de la base de datos, se ha modificado el fichero `game.py` para que se pueda llamar por línea de comandos. Por consiguiente, queda del siguiente modo:

```
python game.py [rule_name] [win_dimensions] [univserse_dimensions]
```

siendo:

- `rule_name` la regla que se ejecutará en el universo. Tiene que existir en la base de datos con el mismo nombre.
- `win_dimensions` las dimensiones de la pantalla.
- `iterations` el número de células que habrá por cada lado en el universo.

Este último se usará para analizar visualmente los resultados obtenidos por el algoritmo y comprobar que, efectivamente, se generan patrones dinámicos.

Capítulo 5

Resultados

En este capítulo se encuentran documentadas las reglas que han sido generadas por el algoritmo, poniendo especial énfasis en las que se generan patrones de tipo *spaceships* (ver apartado 3.2.3). Para facilitar la legibilidad de los resultados se va a usar la siguiente expresión para representar una regla:

$$(E_bE_h/F_bF_h)\{C_n\}_{0\leq n<512},$$

donde (E_bE_h/F_bF_h) es la regla del Espacio de Bays desde la que se parte y C_n todos aquellos contextos en los que difieren la original con el resultado. Se trata de una variación del formato usado en el artículo *Research of a Cellular Automaton Simulating Logic Gates by Evolutionary Algorithms* [1].

Para estructurarlo de alguna forma se agruparán las reglas resultantes según su regla inicial del Espacio de Bays. Cabe mencionar que ninguna de las reglas iniciales utilizadas genera patrones de tipo *spaceship*.

5.1. Regla inicial 12/33

5.1.1. Resultado 1

Notación de la regla: (12/33){3, 30, 111, 146, 196, 202, 250, 274, 322, 350, 356, 399, 424, 425, 432, 448, 460, 461, 492, 498}

5.1.1.1. *Still lifes*

Se han encontrado los siguientes patrones estáticos:

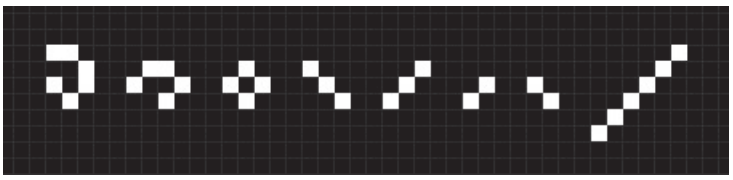

Figura 5.1: Patrones estáticos

Como se observa, cualquier celda que tenga en sus diagonales otra célula viva forma un patrón estático.

5.1.1.2. *Oscillators*

El único oscilador encontrado es el de la la figura 5.2 con un periodo 2.

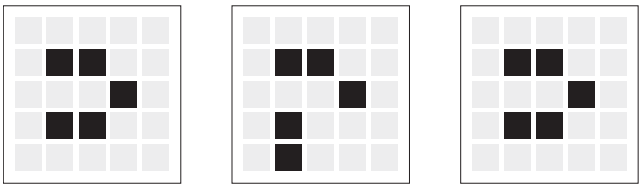

Figura 5.2: Osciladores de la regla generada por el algoritmo.

5.1.1.3. *Spaceships*

Se han encontrado dos *spaceships*. Por una parte, el de la figura 5.3 que tras cuatro iteraciones vuelve a su estado inicial, pero dos células por debajo.

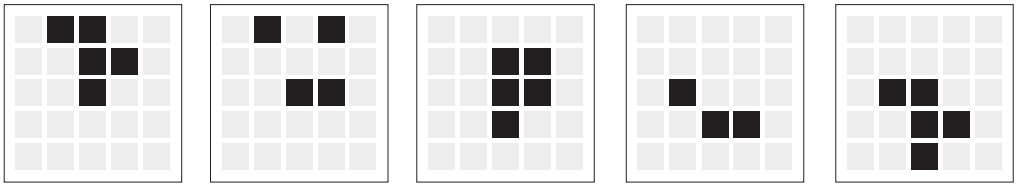

Figura 5.3: Patrón dinámico de la regla generada por el algoritmo.

En segundo lugar, se cuenta con otro patrón similar y que además deja una estela de patrones estáticos. En la figura 5.4 se muestra la ejecución de dicha estructura de izquierda a derecha y de arriba a abajo. Se trata de un patrón donde la disposición inicial se repite tras ocho iteraciones, cuatro celdas por debajo. Se puede observar también cómo deja un rastro de patrones estáticos diagonales.

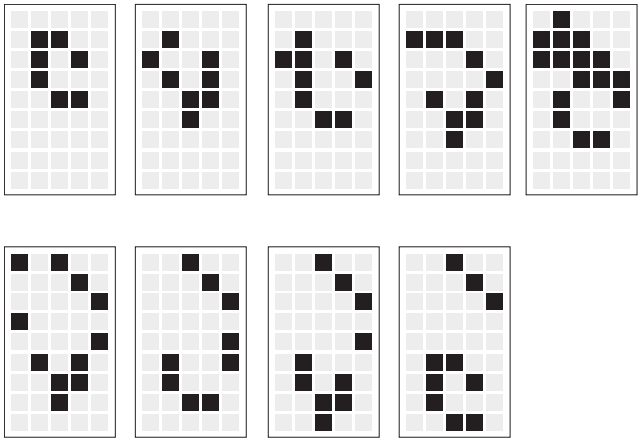

Figura 5.4: Patrón dinámico de la regla generada por el algoritmo.

5.1.2. Resultado 2

Notación de la regla: (12/33){1, 4, 49, 125, 154, 222, 240, 266, 268, 317, 344, 398, 399, 420, 434, 448, 452, 467}

5.1.2.1. *Still lifes*

Como se puede ver en la figura 5.5, el resultado comparte algunos patrones estáticos con la regla anterior a excepción de los patrones estáticos que se generaban en la figura 5.4.

Figura 5.5: Patrones estáticos que se generan.

5.1.2.2. *Oscillators*

El único oscilador encontrado es el de la la figura 5.6 con un periodo de 2.

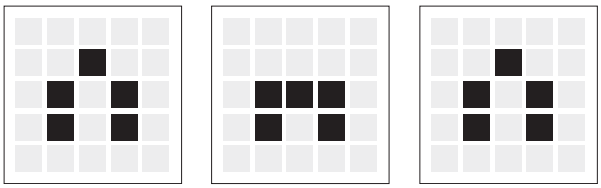

Figura 5.6: Osciladores de la regla generada por el algoritmo.

5.1.2.3. *Spaceships*

En este caso solo se ha encontrado patrón dinámico de la figura 5.7. No obstante, cabe destacar que es similar al primero encontrado en la regla anterior. En este caso tiene periodo seis en el que la posición inicial se repite tres células por debajo. Curiosamente, las iteraciones 1, 2, 5 y 6 aparecían en la figura 5.3 pero invertidas horizontalmente.

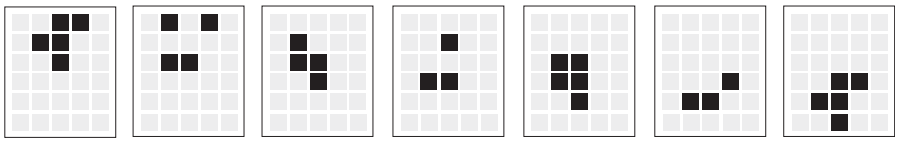

Figura 5.7: Patrón dinámico de la regla generada por el algoritmo.

5.2. Regla inicial 23/44

5.2.1. Resultado 3

Notación de la regla: (23/44){56, 167, 208, 405, 503}

5.2.1.1. *Still lifes*

En la figura 5.8 se muestran algunos de los patrones estáticos encontrados.

Figura 5.8: Patrones estáticos

5.2.1.2. *Oscillators*

Se han observado tres tipos de osciladores con dicha regla. El primero, en la figura 5.9 formado por la composición de tres patrones estáticos y con periodo 2.

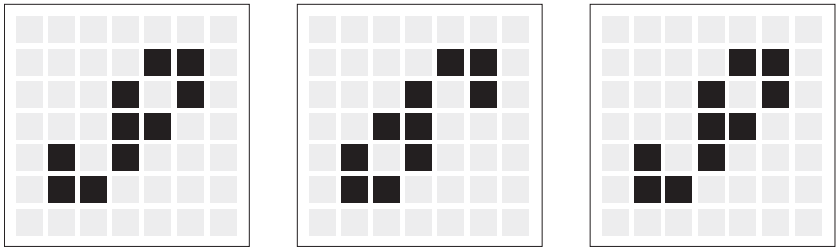

Figura 5.9: Oscilador encontrado con la regla actual

Por otra parte, se tienen los dos que se muestran en la figura 5.10. Lo interesante de estos dos casos es que, además de repetir su configuración inicial, generan continuamente una celda viva que se extiende hacia la izquierda. Por lo tanto, son patrones con la capacidad de autorreplicación.

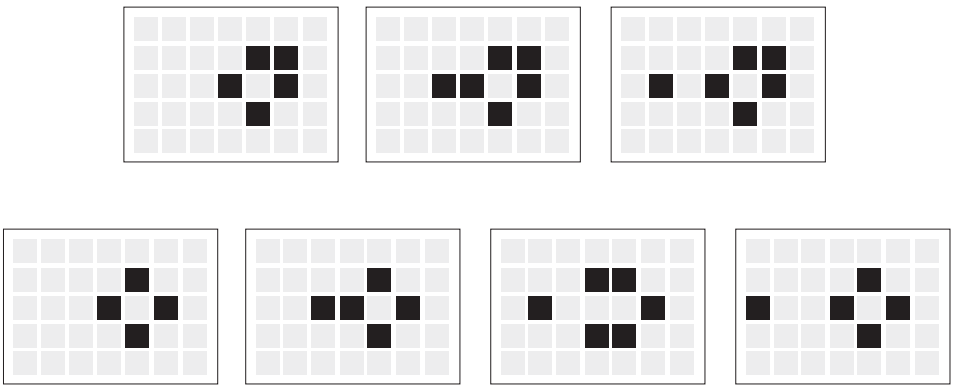

Figura 5.10: Oscilador encontrados, con periodo 3 y 4 respectivamente.

5.2.1.3. *Spaceships*

Esta regla cuenta con una gran cantidad de tipos de patrones dinámicos, muchos de ellos con estructuras similares. A continuación se muestran los más comunes.

El primero de ellos, figura 5.11, cuenta con un periodo de 3 en el que se desplaza tres células. Como se puede observar en el resto de patrones dinámicos, todos se desplazan una célula por iteración, por lo que realmente la regla genera *spaceships* de velocidad máxima.

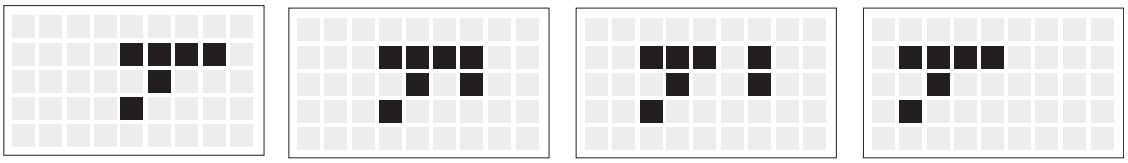

Figura 5.11: Patrón dinámico encontrado.

En el caso de la figura 5.12, se desplaza cuatro celdas cada cuatro iteraciones.

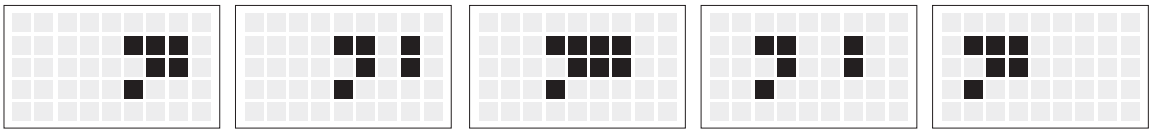

Figura 5.12: Patrón dinámico encontrado.

Por último, otro patrón interesante es el de la figura 5.13. Dicho patrón genera una estela de patrones estáticos cada cuatro iteraciones.

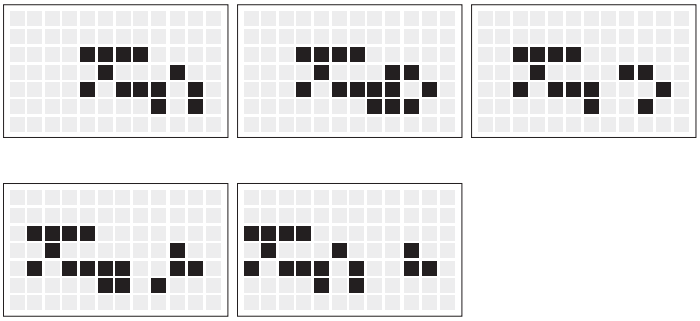

Figura 5.13: Patrón dinámico encontrado.

5.3. Regla inicial 24/44

5.3.1. Resultado 4

Notación de la regla: (24/44){206, 220, 227, 320, 358, 435, 451, 505}

5.3.1.1. *Still lifes*

En la figura 5.14 se representan algunos de los patrones estáticos más comunes. La regla en cuestión genera una gran variedad de ellos.

5.3.1.2. *Oscillators*

En esta regla se forman una gran cantidad de osciladores de periodo 2, como los de las figuras 5.15 y 5.16.

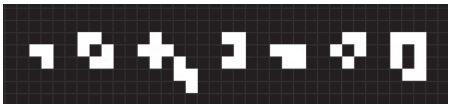

Figura 5.14: Algunos de los patrones estáticos generados.

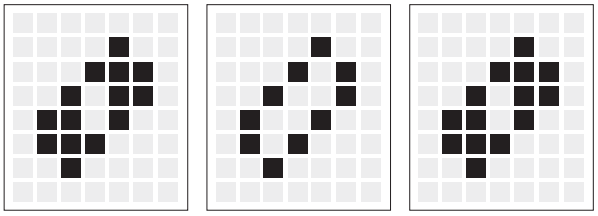

Figura 5.15: Oscilador de periodo 2.

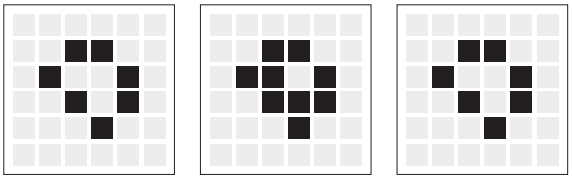

Figura 5.16: Oscilador de periodo 2.

También se ha encontrado un único oscilador, el de la figura 5.17, con un periodo de 3.

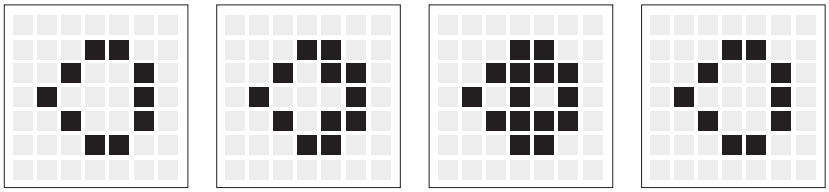

Figura 5.17: Oscilador de periodo 3.

5.3.1.3. *Spaceships*

El único patrón de este tipo que se ha encontrado es el de la figura 5.18 de periodo 2 y que sea desplaza dos celdas a la derecha.

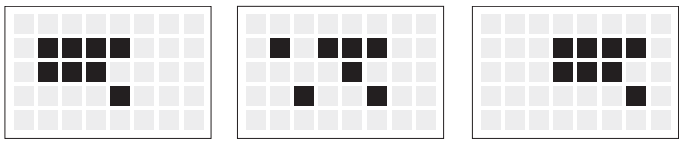

Figura 5.18: Patrón dinámico de periodo 2 encontrado en el resultado actual.

5.3.2. Resultado 5

Notación de la regla: (24/44){42, 48, 57, 150, 177, 210, 249, 254, 277, 293, 328, 334, 351, 391, 423, 439, 457, 473, 503, 504}

5.3.2.1. *Still lifes*

En la figura 5.19 se representan los patrones estáticos que más se generan. Se generan muchos de ellos, pero en su gran mayoría son pequeños.

Figura 5.19: Algunos de los patrones estáticos generados.

5.3.2.2. *Oscillators*

Al contrario que los resultados anteriores, la mayoría de osciladores que se generan son de periodos mayores a 4. La figura 5.20 es un ejemplo de uno de periodo 6.

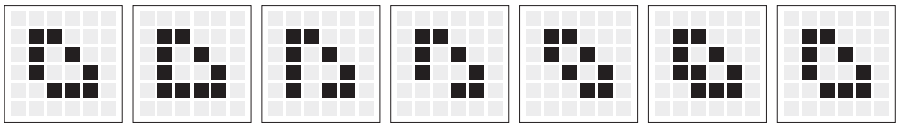

Figura 5.20: Patrón de tipo *oscillator* de periodo 6.

En cuanto a la figura 5.21, podemos observar que además de repetir su configuración inicial crea un patrón dinámico tras seis iteraciones.

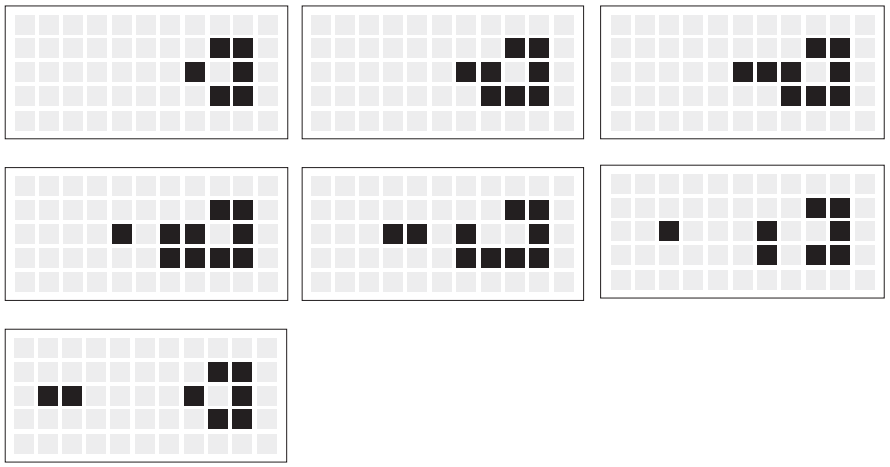

Figura 5.21: De izquierda a derecha y de arriba a abajo, patrón *oscillator* de periodo 6.

5.3.2.3. *Spaceships*

Esta configuración cuenta con una gran variedad de patrones dinámicos, a continuación se muestran los tres más comunes. En primer lugar, en la figura 5.22 se observa un patrón de periodo 3 en el que recorre tres celdas hacia la izquierda.

En segundo lugar, en la figura 5.23 se tiene otra *spaceship* formada por muy pocas células vivas. En concreto, partiendo de dos celdas vivas y tras dos iteraciones se repite su estado inicial dos posiciones a la izquierda.

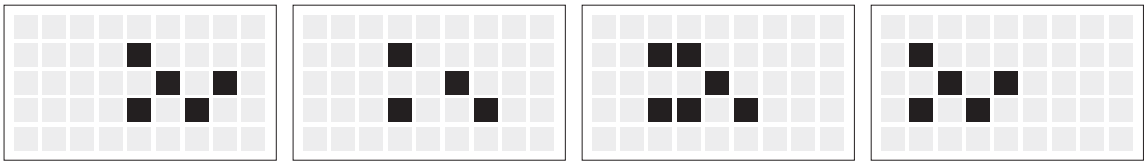

Figura 5.22: Patrón de tipo *spacehip* de periodo 3.

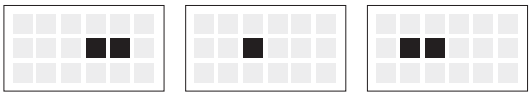

Figura 5.23: Patrón de tipo *spacehip* de periodo 2.

Por último, en la figura 5.24 se encuentra otra estructura en el que, en este caso, tras cuatro iteraciones se desplaza cuatro celdas a la izquierda.

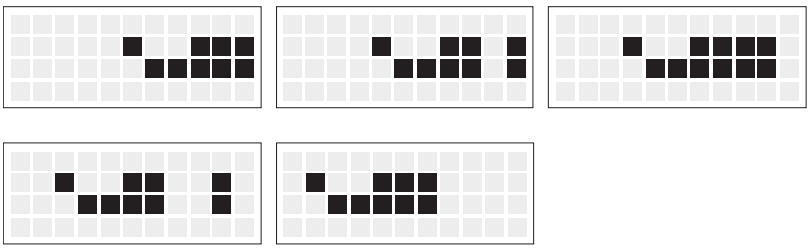

Figura 5.24: Patrón de tipo *spacehip* de periodo 4.

5.4. Conclusiones y posibles mejoras

En conclusión, se observa que el algoritmo efectivamente consigue encontrar reglas con patrones de tipo *spaceship*. No obstante, cabe destacar que no todos y cada uno de los resultados que se generan son correctos ya que algunos consiguen puntuar bien "engañando" al algoritmo al tener un buen valor de *fitness*. Por esta razón, una posible mejora sería revisar las conclusiones descritas en el apartado 4.2.2 y añadir nuevos parámetros de control que eviten dichos resultados.

Por otra parte, aunque el código actual está optimizado respecto a las primeras versiones, otro paso a seguir sería el intentar mejorar los tiempos de ejecución. Actualmente, cada simulación que el algoritmo realiza para calcular el *fitness* de cada individuo de la población dura alrededor de seis segundos. Teniendo 50 individuos se puede calcular que cada iteración del algoritmo (ver figura 4.1) tarda unos 300 segundos. Como realmente el algoritmo solo trabaja con matrices y vectores de enteros, una posible solución podría ser pasar el código al lenguaje de programación C ya que es bastante más rápido que Python. Se perdería la facilidad con la que se pueden llevar a cabo ciertas tareas, pero con el tiempo de desarrollo necesario se podría obtener un programa que hiciese exactamente lo mismo que el actual.

Finalmente, una vez corregidos los problemas anteriores, se podría estudiar el desarrollo de otro algoritmo genético que busque en este caso reglas capaces de generar estructuras análogas al *Gosper Glider Gun* (ver apartado 3.2.4). Para que un autómata pueda imitar el comportamiento de un ordenador se tienen que poder formar estructuras dinámicas

como las vistas en este capítulo y, además, tiene que tener un patrón capaz de generar dichas estructuras de forma continua e indefinida. Con el alcance actual del trabajo se tienen diferentes reglas que cumplen la primera condición, por lo que podrían ser posibles miembros de la población inicial del futuro algoritmo en cuestión.

Capítulo 6

Análisis de impacto

Por último, en cuanto al análisis de impacto del trabajo se puede decir que es puramente cultural. Si bien es cierto que los autómatas celulares cuentan con numerosos usos como los mencionados en el segundo capítulo, este proyecto en concreto sirve como estudio y exposición del Juego de la Vida para aquellas personas que no estén familiarizadas con el término. Además, también se añade la conexión con los algoritmos evolutivos lo que proporciona más valor educativo al ser una presentación a ciertos métodos de optimización heurística. En definitiva, los beneficios culturales y académicos del proyecto tendrán más peso que otros aspectos como los sociales, empresariales, medioambientales, etc.

Bibliografía

[1] E. Sapin, O. Bailleux, and J.-J. Chabrier, "Research of a cellular automaton simulating logic gates by evolutionary algorithms," 04 2003, pp. 414–423.

[2] Ian07. (2018) Hexagonal tiling. [Online]. Available: https://conwaylife.com/wiki/Hexagonal_tiling

[3] D. S. Suzana Dragićević, "A gis-based irregular cellular automata model of land-use change," *Environment and Planning B: Planning and Design*, vol. 34, no. 4, pp. 708–724, 2007.

[4] C.Bays, "Candidates for th e game of life in three dimensions," *Complex Systems*, vol. 1, pp. 373–400, 1987.

[5] G. Z.-Y. Ke-Ping Li, "Cellular automation model of traffic flow based on the car-following model," *Chinese Physics Letters*, vol. 21, no. 11, 2004.

[6] E. D. R. Slimi, S. El Yacoubi and S. Gourbière, "A cellular automata model for chagas disease," *Applied Mathematical Modelling*, vol. 33, no. 2, pp. 1072–1085, 2009.

[7] C. C. D. Joana Gouveia Freire, "Using cellular automata to simulate wildfire propagation and to assist in fire management," *Natural Hazards and Earth System Sciences*, vol. 19, no. 1, pp. 169–179, 2019.

[8] N. S. Damien Regnault and Éric Thierry, "On the analysis of "simple" 2d stochastic cellular automata," *Discrete Mathematics & Theoretical Computer Science*, vol. 12, no. 2, pp. 263–294, 2010.

[9] C. Bays, *Game of Life Cellular Automata*. Springer, 2010.

[10] S. Wolfram, *A new Kind of Science*. Wolfram Research, 2002.

[11] Conway's game of life viewer. [Online]. Available: https://copy.sh/life

[12] Lifewiki. [Online]. Available: https://conwaylife.com/wiki/Main_Page

[13] N. Carlini. Digital logic gates on conway's game of life. [Online]. Available: https://nicholas.carlini.com/writing/2020/digital-logic-game-of-life.html

[14] Time complexity. [Online]. Available: https://wiki.python.org/moin/TimeComplexity

Este documento esta firmado por

	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Thu Jun 02 20:49:14 CEST 2022
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)